

Aggressive Performance Optimizations for 3D Graphics

Haim Barad (organizer),
Eric Haines, Dean Macri,
Kim Pallister and Alex Klimovitski

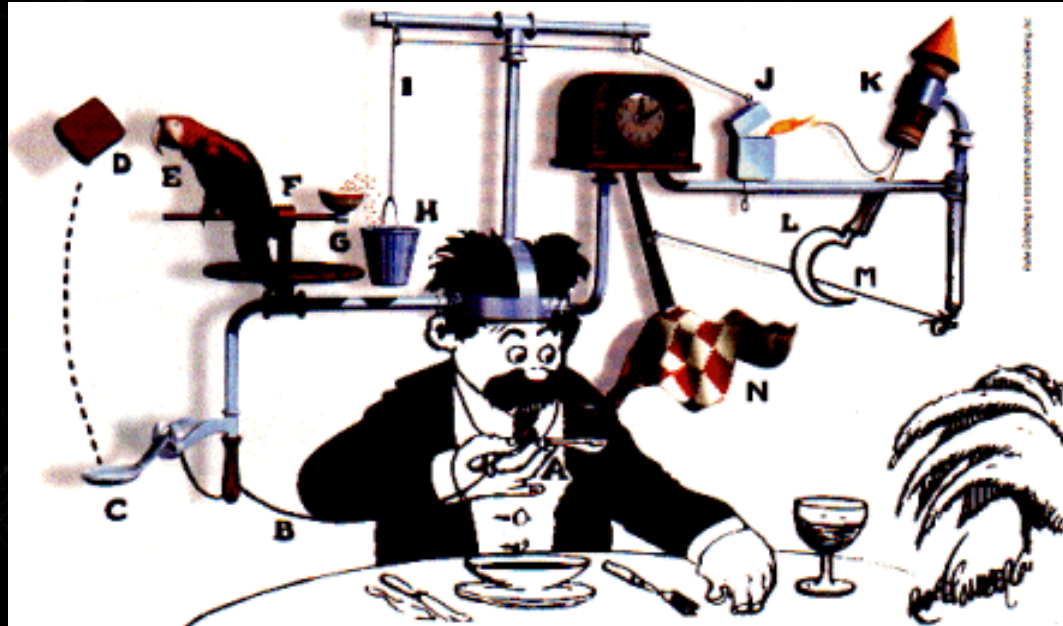
Concept #1: Aggressive Performance Optimizations

“Dem’s powerful words...”

Could also be called “Aggressively Pursuing Optimal Performance”

- **Aggressive** – use all methods, knowledge & tools available to achieve...
- **Pursuing** – it’s an iterative process
- **Optimal** – there doesn’t exist anything better...
- **Performance** – make it FAST!

Most SW isn't this bad, but...



RAISING SPOON TO MOUTH

(A) pulls string (B), thereby jerking ladle (C), which throws cracker (D) past parrot (E). Parrot jumps after cracker, and perch (F) tilts, upsetting seeds (G) into pail (H). Extra weight in pail pulls cord (I), which opens and lights automatic cigar lighter (J), setting off skyrocket (K), which causes sickle (L) to cut string (M) and allows pendulum with attached napkin (N) to swing back and forth, thereby wiping off your chin.

Concept #2: “Bad software slows down good hardware...”

Algorithmic

- Avoid unnecessary work
- Do necessary work in a smart way

Architectural

- Know your target platform

Coding

- Use code wisely

Levels of optimizations

Algorithmic

- Biggest returns by doing it SMART!
- Must be done first

Architectural

- Leverage characteristics & capabilities of the platform

Coding

- Write code, analyze, recode, analyze... squeeze!

Concepts are widely applicable

Not just to Intel* architecture

- Algorithmic
- Architectural
- SIMD
- Memory optimizations

Applicable to all architectures & platforms!

- From server to embedded CPUs...

*All brands and names are the property of their respective owners.

Schedule

8:30 Welcome and Overview - Barad

8:45 3D Algorithmic Optimizations - Haines

10:00 Morning Break

10:15 Implementations for Bandwidth Reduction - Pallister

11:30 Architecture & Microarchitectural Issues - Barad

12:00 Lunch

1:30 Optimizations Lab - Klimovitski

3:00 Afternoon Break

3:15 Analysis Tools & Lab - Macri

4:25 3D Optimizations for PDAs - Barad

4:55 Summary, Wrap Up, Questions, Feedback - All

Skills needed for labs...

Not too much is needed

- Basic understanding of C++
- Some experience with building and analyzing applications

Course materials and updates are online

- <http://Optimizations.org>

Who's who...

Haim Barad - Intel

- barad@acm.org

Eric Haines - Autodesk

- erich@acm.org

Dean Macri - Intel

- dean.p.macri@intel.com

Kim Pallister - Intel

- kim.pallister@intel.com

Alex Klimovitski - Intel

- Alex.klimovitski@intel.com

3D Algorithmic Optimizations

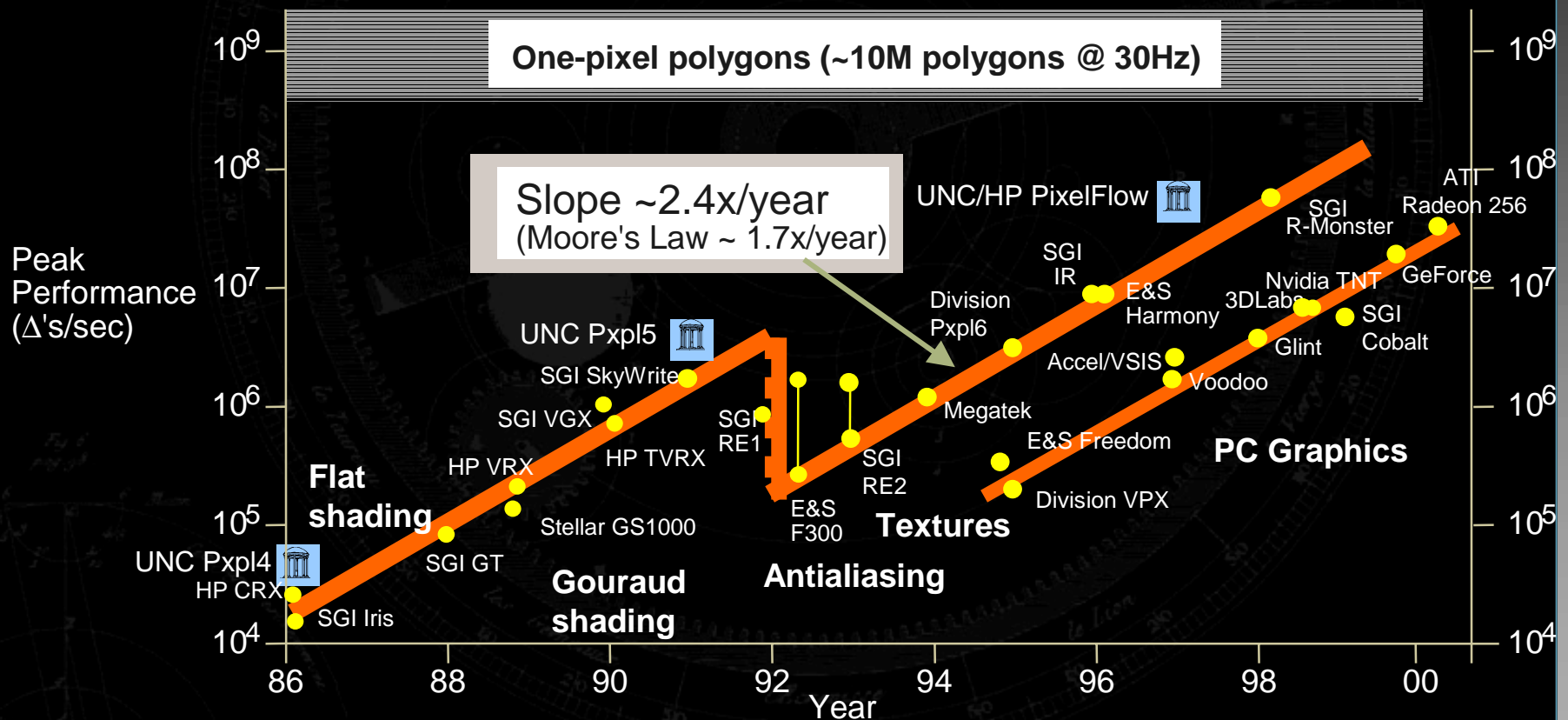
Eric Haines
Autodesk, Inc.
erich@acm.org

Trends

Moore's Law: 1.7x increase in CPU speed per year

Graphics accelerators are 2x to 4x (Poulton at UNC-CH gives 2.4x per year since 1986)

Faster than Moore's Law



Graph courtesy of Professor John Poulton

Why care about faster algorithms for CG?

Simple answer: never too much speed!

- Screen resolution (4000 x 2500 ?)
- Frame rate (72 Hz)
- Realism (photorealism might be the upper limit...)
- Scene complexity (no upper limit) !

Algorithm or Optimization?

As time goes on, the algorithm is more important than the optimization...

Example: Compare BubbleSort / QuickSort

- BubbleSort will never win (well, almost never...) no matter how much you optimize the code

However, then code and memory optimizations can make Qsort even faster!

Application Side Algorithms

Push less across the bus:

- Culling techniques
- Share vertices: strips, vertex buffers
- Compress geometric description (Deering)
- Use other descriptions (images, Beziers, subdivision surfaces)
- Send simplified models (same amount of fill, less vertices)

Culling Techniques

“To cull” means “to select from group”

In graphics context: do not process data that will not contribute to the final image

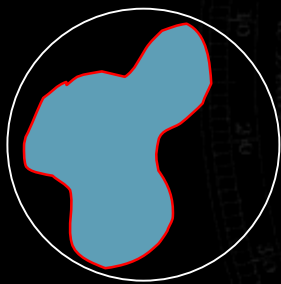
The “group” is the entire scene, and the selection is a subset of the scene that we do not consider to contribute

Culling: Overview

- Backface culling
- Hierarchical view-frustum culling
- Portal culling
- Detail culling
- Occlusion culling

Culling Examples

view frustum

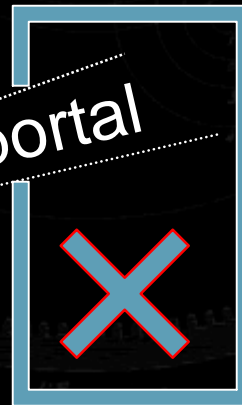


■ detail

backface



portal



occlusion



Backface Culling

Simple technique to discard polygons that faces away from the viewer

Can be used for:

- closed surface (example: sphere)
- or whenever we know that the backfaces never should be seen (example: walls in a room)

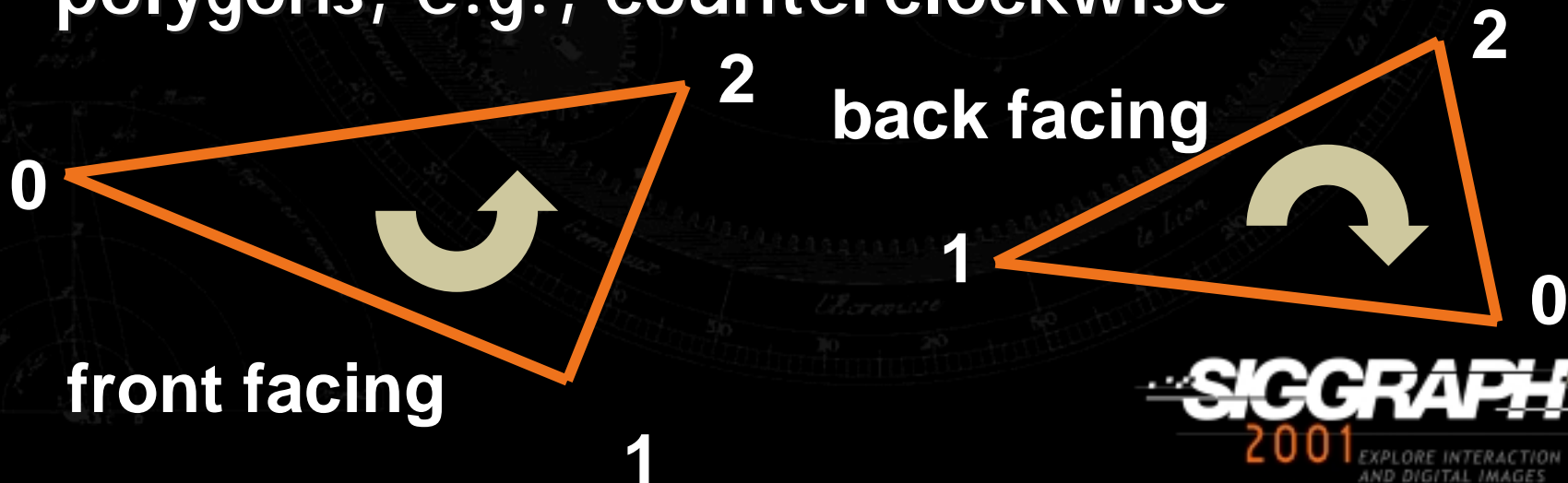
Backface culling (cont'd)

Often implemented for you in the API

OpenGL: `glCullFace(GL_BACK);`

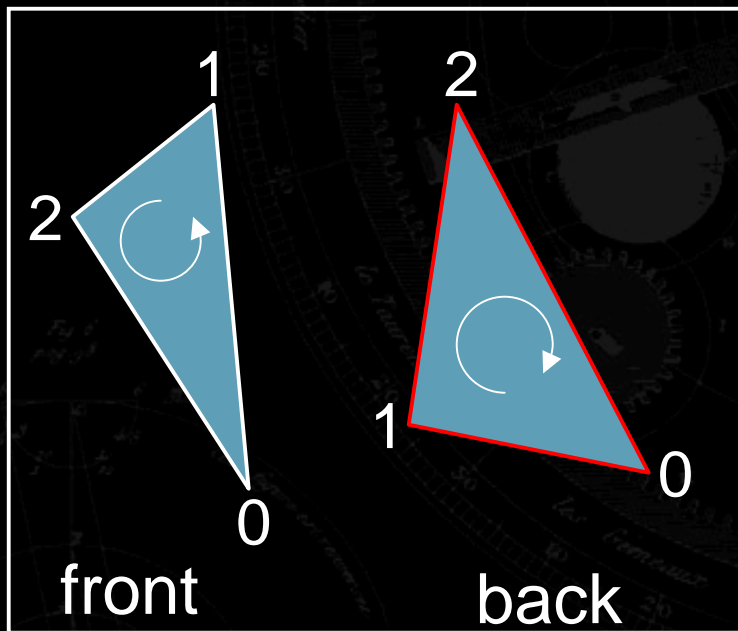
How to determine what faces away?

First, must have consistently oriented polygons, e.g., counterclockwise

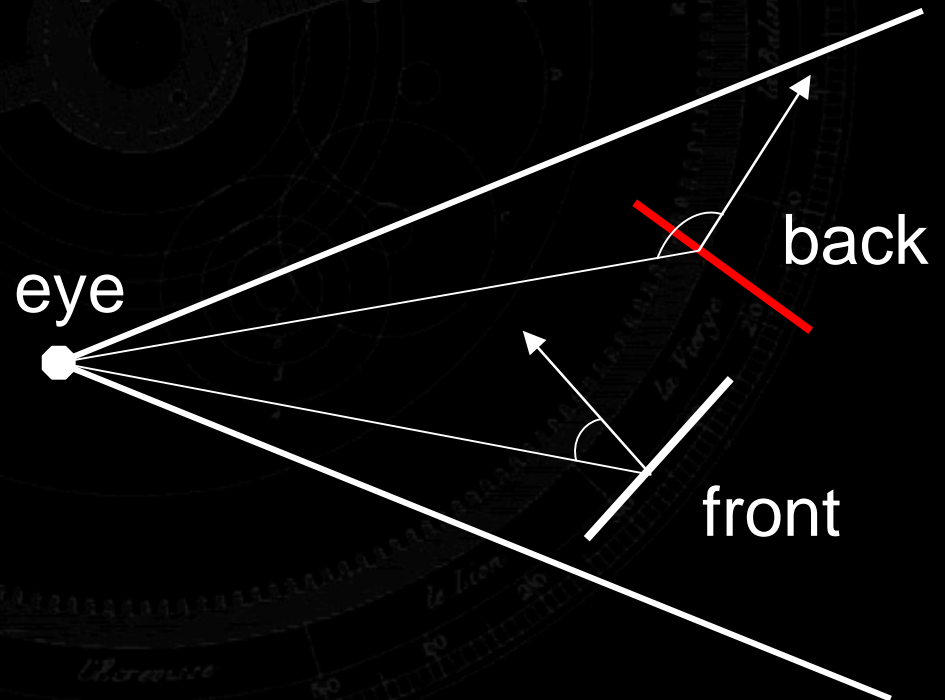


How to cull backfaces

Two methods (screen space, eye space)



screen space



View-Frustum Culling

Bound every “natural” group of primitives by a simple volume (e.g., sphere, box)

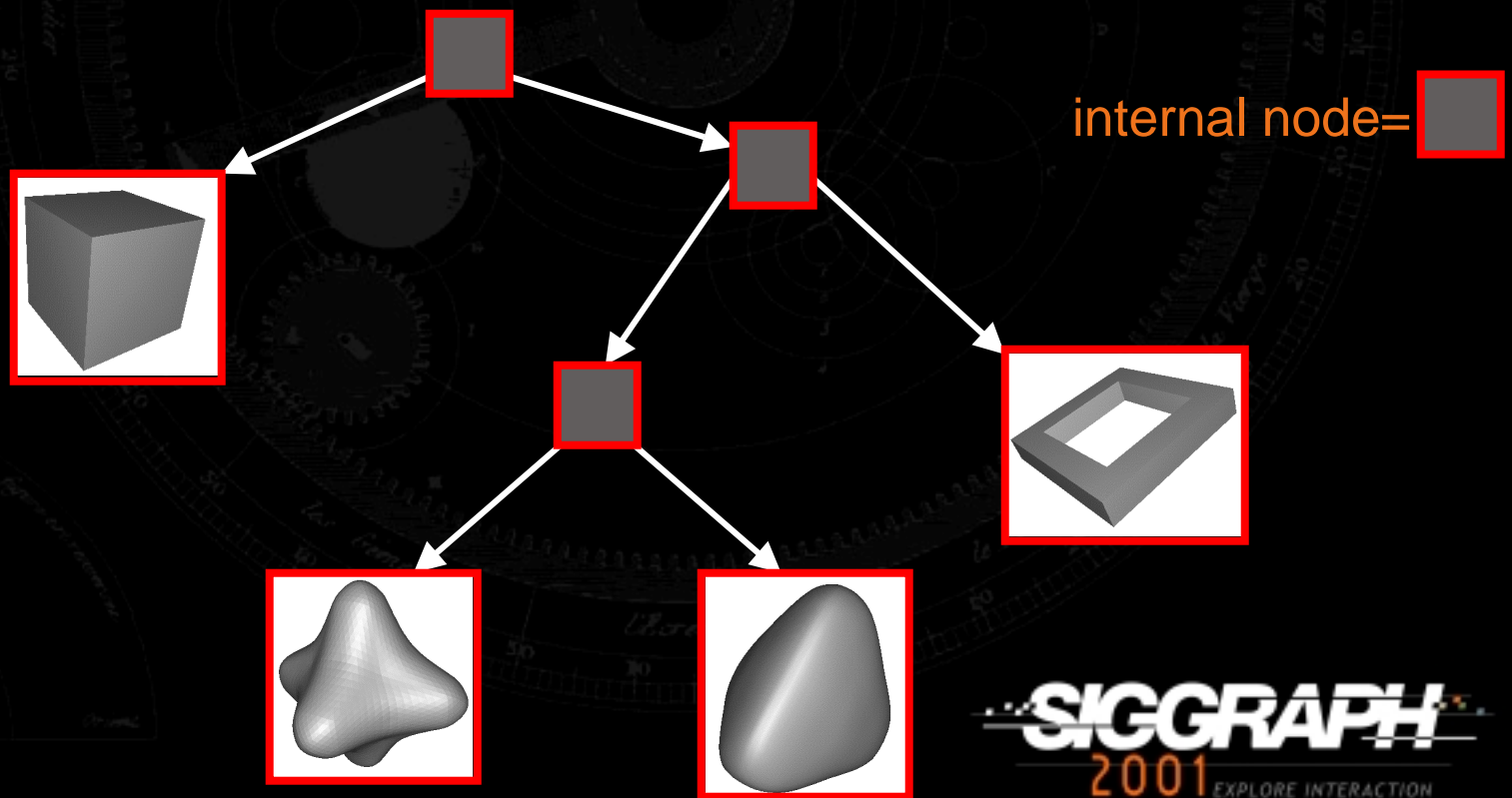
If a bounding volume (BV) is outside the view frustum, then the entire contents of that BV is also outside (not visible)

In the APP stage: avoid further processing of such BV's and their contents

The Scene Graph

DAG - directed acyclic graph

- Simply an n -ary tree without loops



Can we accelerate view frustum culling further?

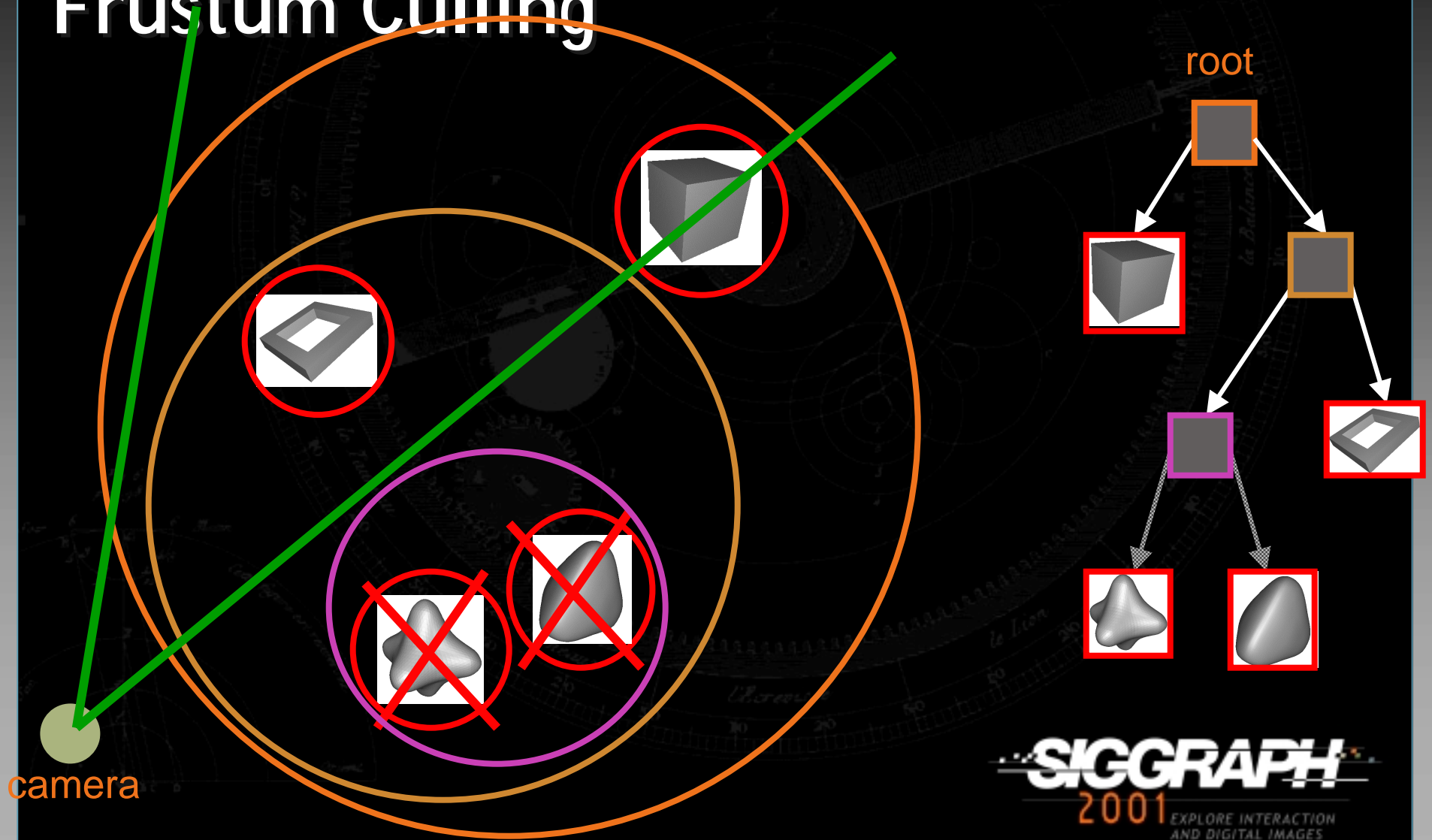
Do what we always do in graphics...

Use hierarchies [Clark76] !

Build some kind of tree hierarchically

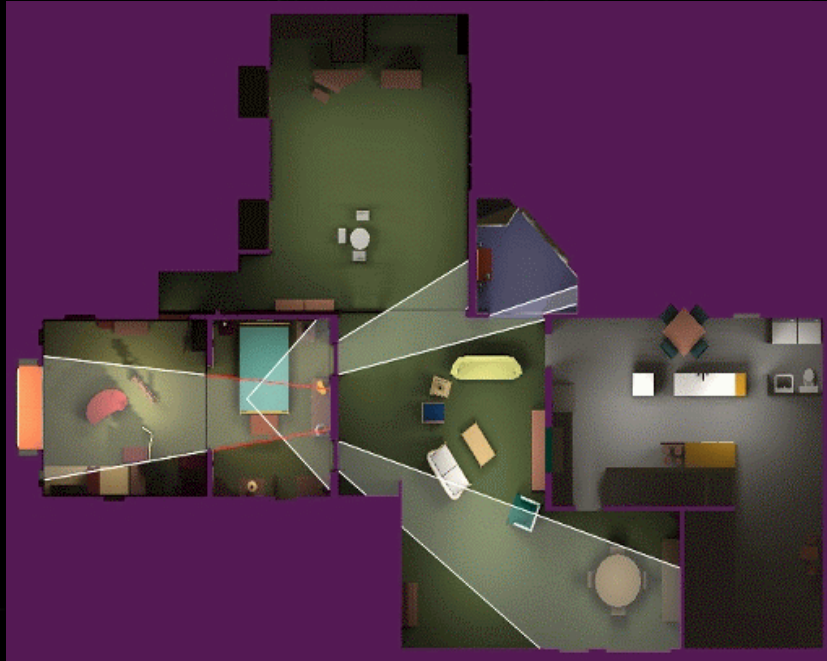
- Could use the existing scene graph
- Could build a more optimal one
 - Tradeoff: speed vs. editability

Example of Hierarchical View Frustum Culling



Portal Culling

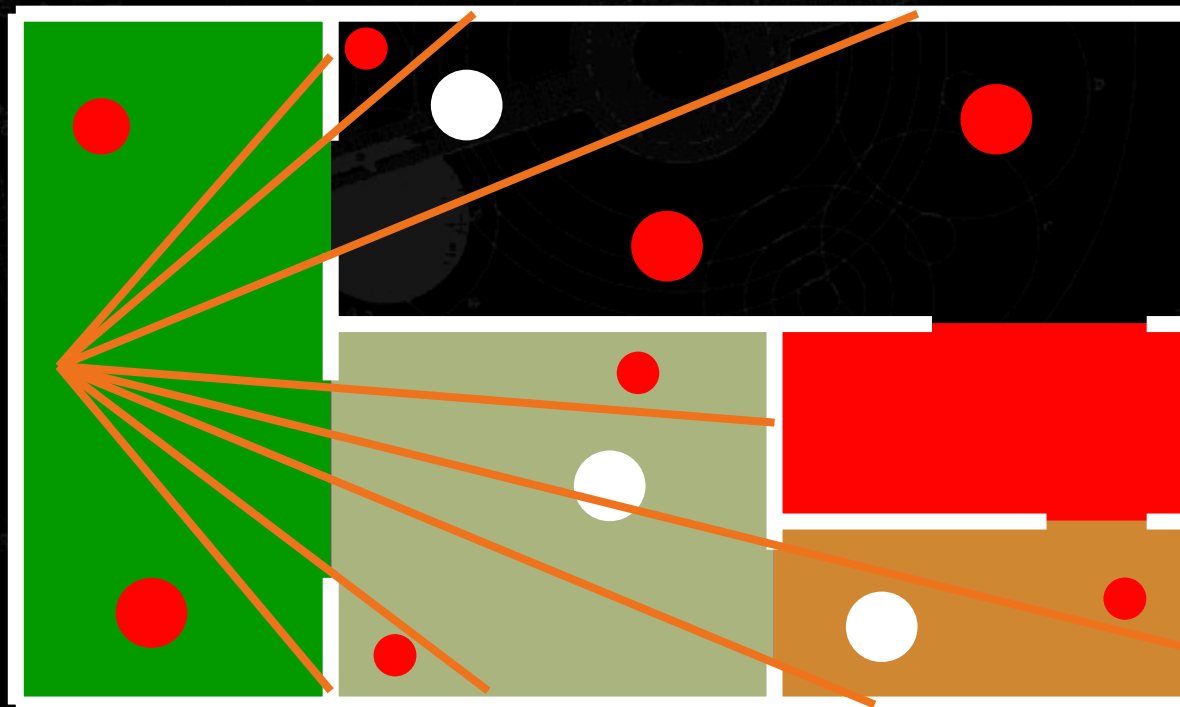
Images courtesy of David P. Luebke and Chris Georges



Culled 20-50% of the polys in view
SU: from slightly better to 10x

Portal culling example

In a building from above
Circles are objects to be rendered



Portal Culling Algorithm

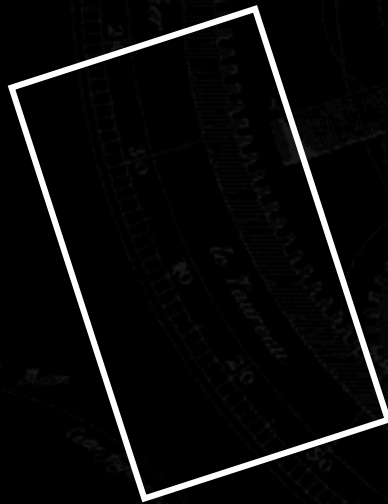
Divide into cells with portals (build graph)

For each frame:

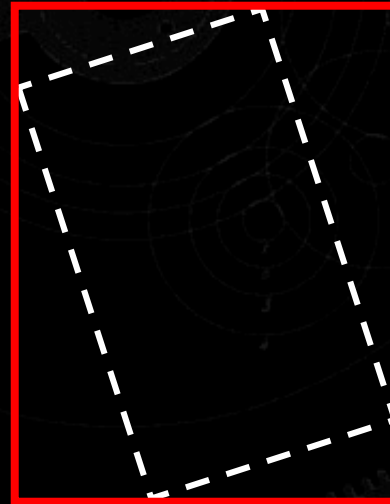
- Locate cell of viewer and init 2D AABB to whole screen
- * Render current cell with VF cull w.r.to AABB
- Traverse to closest cells (through portals)
- Intersection of AABB & AABB of traversed portal
- Goto *

Portal Overestimation

To simplify:



actual portal



overestimated portal

Portal Culling Algorithm

When to exit:

- When the current AABB is empty
- When we do not have enough time to render a cell (“far away” from the viewer)

Also: mark rendered objects

Source (for Performer):

<http://www.cs.virginia.edu/~luebke/>

Demo: Surrender's *Umbra* Building



Q W E

← Movement

A S D

(and "spacebar" for speed)

← →

← Rotation

Right

mouse

← Menus

SIGGRAPH
2001 EXPLORE INTERACTION
AND DIGITAL IMAGES

Detail Culling

Idea: objects whose projected BV occupy less than N pixels are culled

This is an approximative algorithm as the things you cull away may actually contribute to the final image

Advantage: trade-off quality/speed

Example of Detail Culling

Images courtesy of ABB Robotics Product, created by Ulf Assarsson



detail culling OFF

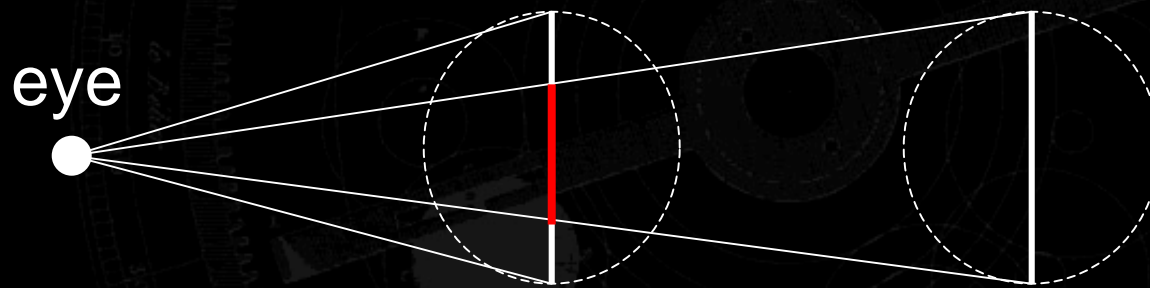


detail culling ON

Not much difference, but 80-400% faster.

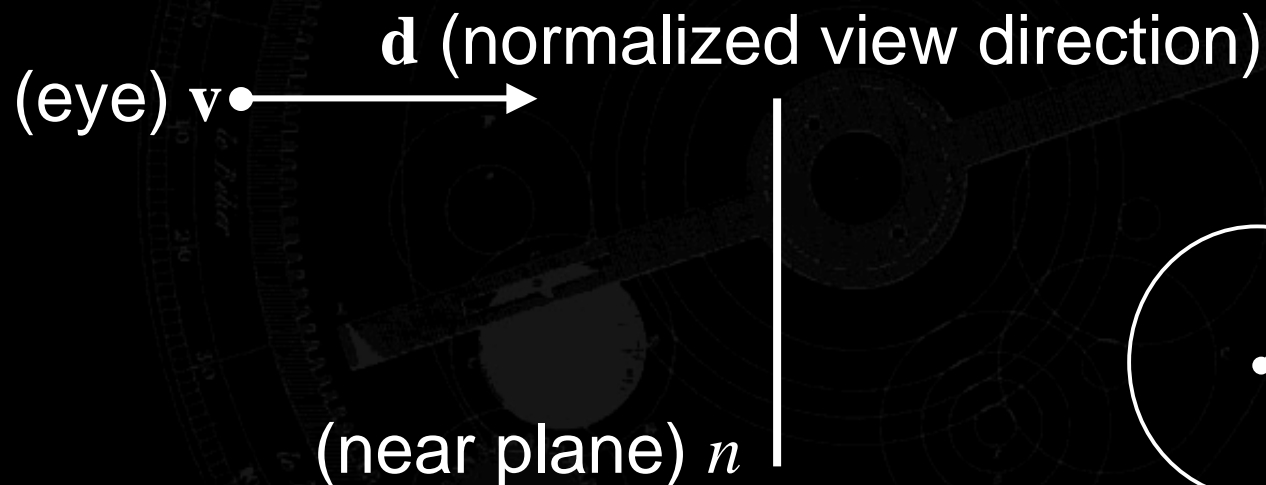
Good when moving

Projection



Projection gets halved when distance is doubled

Projection (cont'd)



$\text{dot}(\mathbf{d}, (\mathbf{c}-\mathbf{v}))$ is distance along \mathbf{d}

$p = nr / \text{dot}(\mathbf{d}, (\mathbf{c}-\mathbf{v}))$ is estimation of projected radius

πp^2 is the area

Quick Hack Occlusion Culling

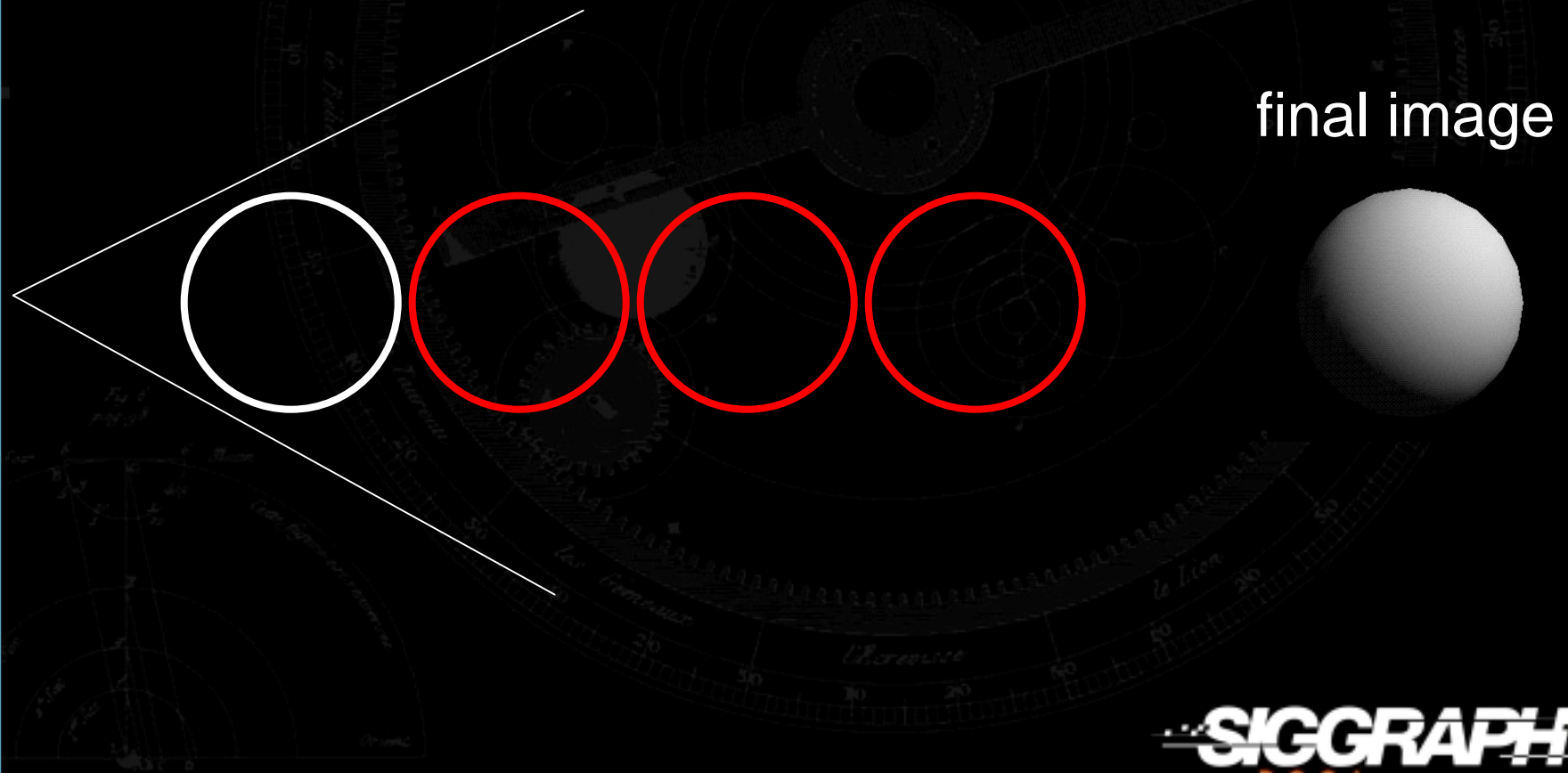
Use fog to fade things out as they get far away - the objects are occluded by the fog, so no longer have to be rendered.

“Real” Occlusion Culling

Main idea: Objects that
lies completely
“behind” another set
of objects can be
culled

We present only one
algorithm here; there
are many schemes

Example



VISUALIZE fx's Occlusion Culling Algorithm

Algorithm (extension to OpenGL):

- Scan convert faces of object, typically bounding box of complex object, but do not write Z
- Get boolean which says if there was a Z-value from scan conversion that was closer than that of the Z-buffer (NVIDIA: get pixels seen count)
- If seen, render complex object

VISUALIZE fx's Occlusion Culling Algorithm (cont'd)

Implications:

- If an object is occluded, then we gain (hopefully) a lot of performance since we only scan convert one Bounding Box (BB) instead of the entire object
- If BB is not occluded, then we have to render the object, and we lose a little performance

Drawing order matters: drawing front-to-back gives more occlusion

Occlusion culling algorithm

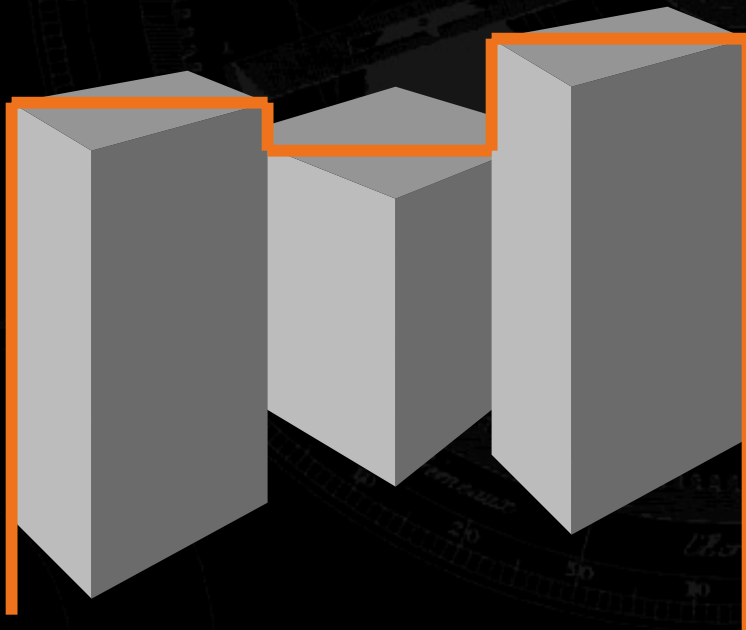
Use some kind of occlusion representation O_R

```
for each object  $g$  do:  
  if( not Occluded( $O_R, g$ ))  
    render( $g$ );  
    update( $O_R, g$ );  
  end;  
end;
```

Occlusion culling algorithm example

Process from front to back

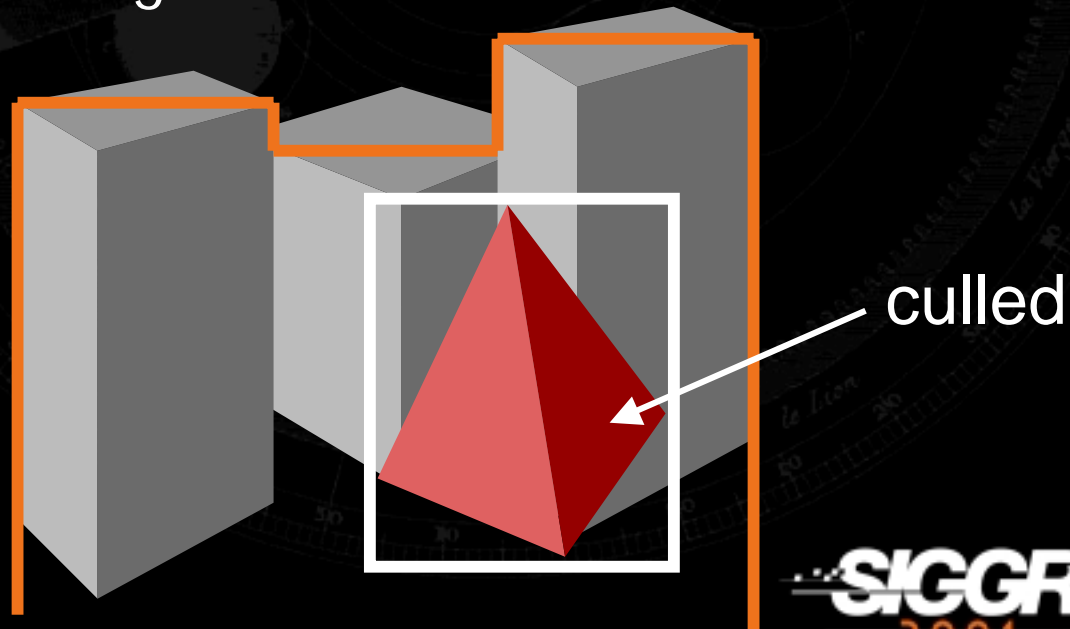
Maintain an occlusion horizon (yellow)



Occlusion culling algorithm example

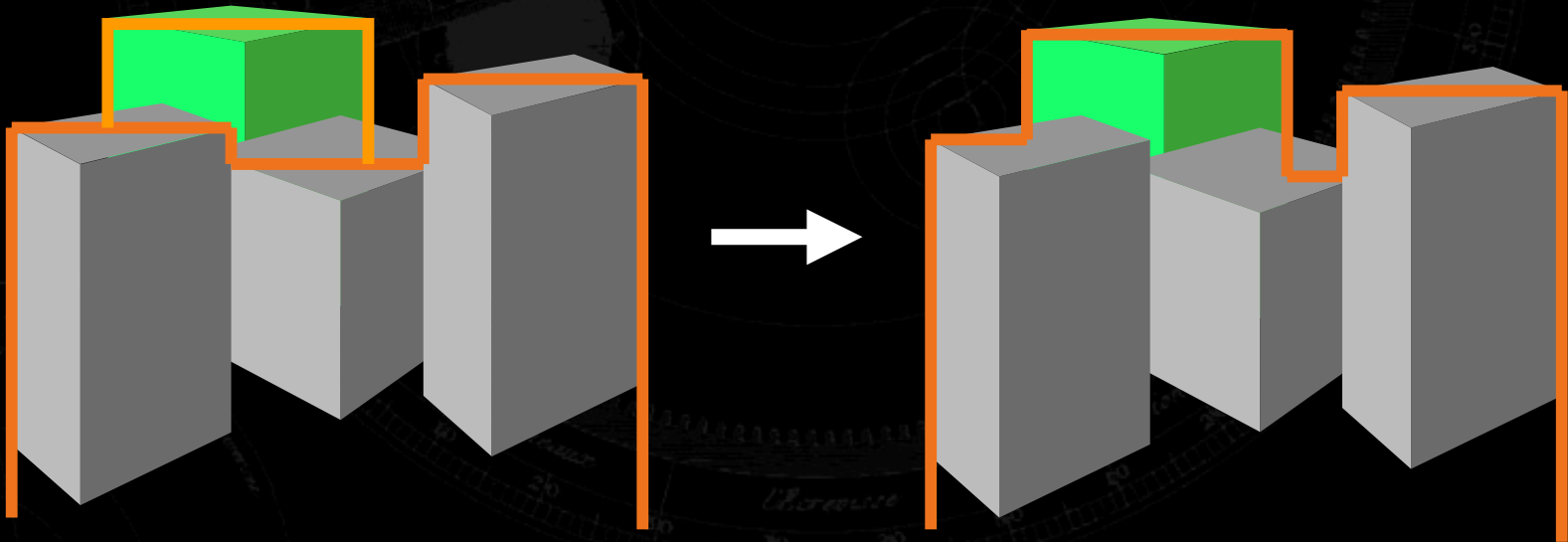
To process tetrahedron (which is behind
grey objects):

- find axis-aligned box of projection
- compare against occlusion horizon

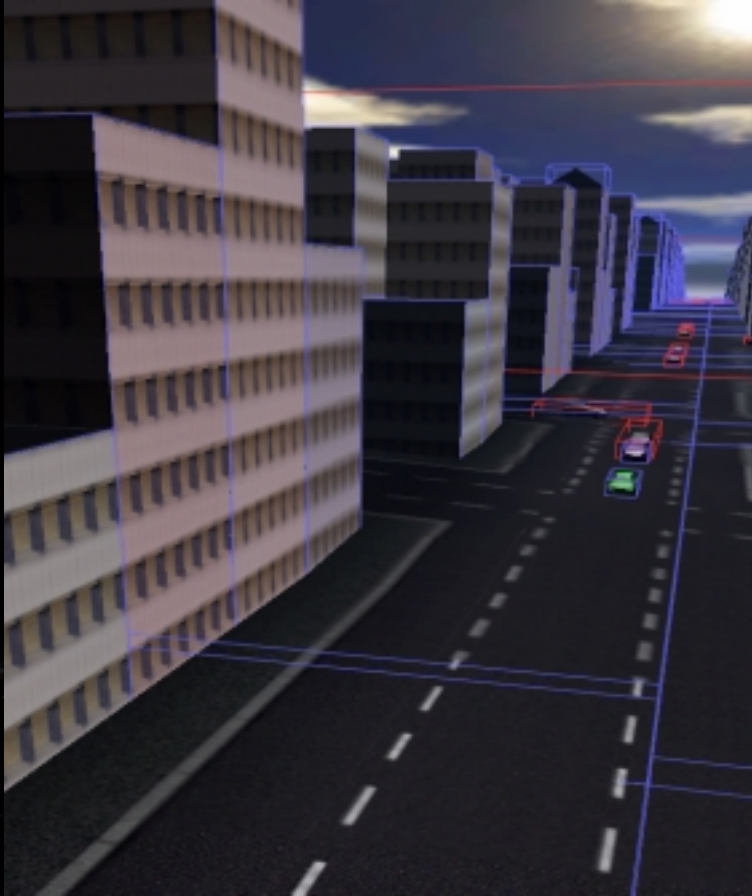


Occlusion culling algorithm example

When an object is considered visible:
Add its “occluding power” to the occlusion representation



Demo: Surrender's *Umbra* Urban



16k buildings, 4k cars

Q W E

← Movement

A S D

(and "spacebar" for speed)

← →

← Rotation

Both

buttons

← Pan

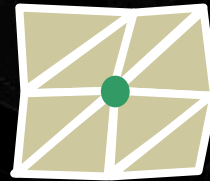
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES

Polygons vs. Images

Which is better to send to the graphics accelerator?

Bandwidth Analysis

Polygonal mesh: each vertex shared by 6 triangles.



Each vertex has a position, normal, and one texture coordinate pair (at least).

Triangle loop itself needs 3 bytes (for a mesh with ≤ 256 vertices)

Bandwidth Analysis (cont.)

Triangle takes:

$3 \times (3 \text{ XYZ floats} + 3 \text{ normal floats} + 2 \text{ UV floats}) / 6 + 3 \text{ bytes for each loop} =$
19 bytes per triangle

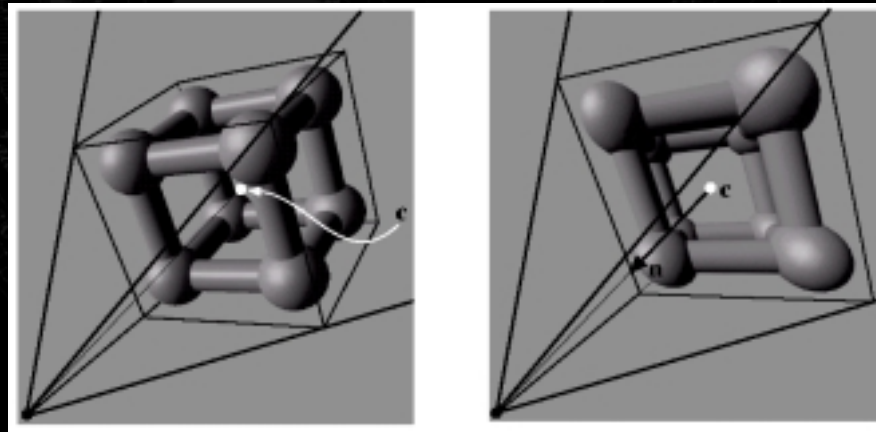
There are 3 bytes per pixel (RGB)

So a triangle must *be visible at* $19/3 \approx 6$ pixels to cost less than sending the pixels.

(idea by Steve Hollasch)

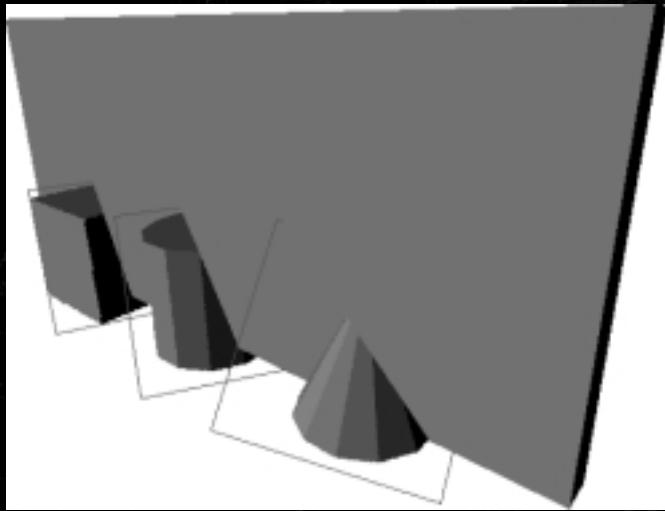
Impostors

For a far away object, render it once and send down the image as a textured quadrilateral with alpha transparency.

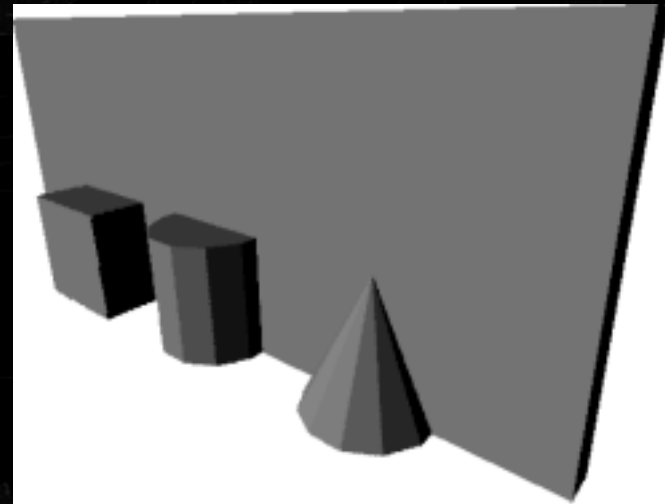


Nailboards

Nailboards are impostors with relative Z depth information stored. Allows overlap:



Imposters



Nailboards

Images courtesy Gernot Schaufler

Simplified Models/LOD

Levels of Detail (LOD): when an object is far away, use a simpler version of it.

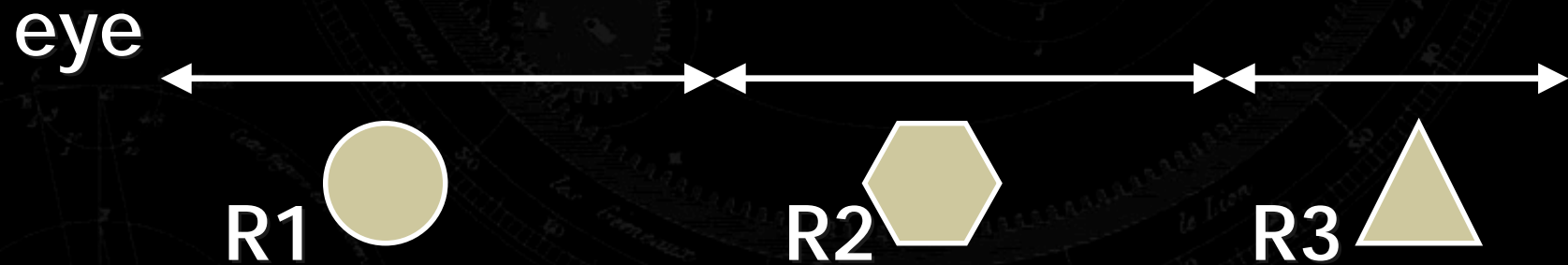
Some types of LODs:

- Discrete LODs
- Alpha LODs
- Geomorph LODs

Discrete LODs

Multiple versions of same model.

Distance or screen size ranges are used:



Level-of-Detail Rendering

Use different levels of detail at different distances from the viewer

More triangles closer to the viewer



LOD rendering

Not much visual difference, but a lot faster



- Use area of projection of BV to select appropriate LOD

Far LOD rendering

When the object is far away, replace with a quad of some color

When the object is *really far away*, do not render it (detail culling)!

Drawbacks of Discrete LODs

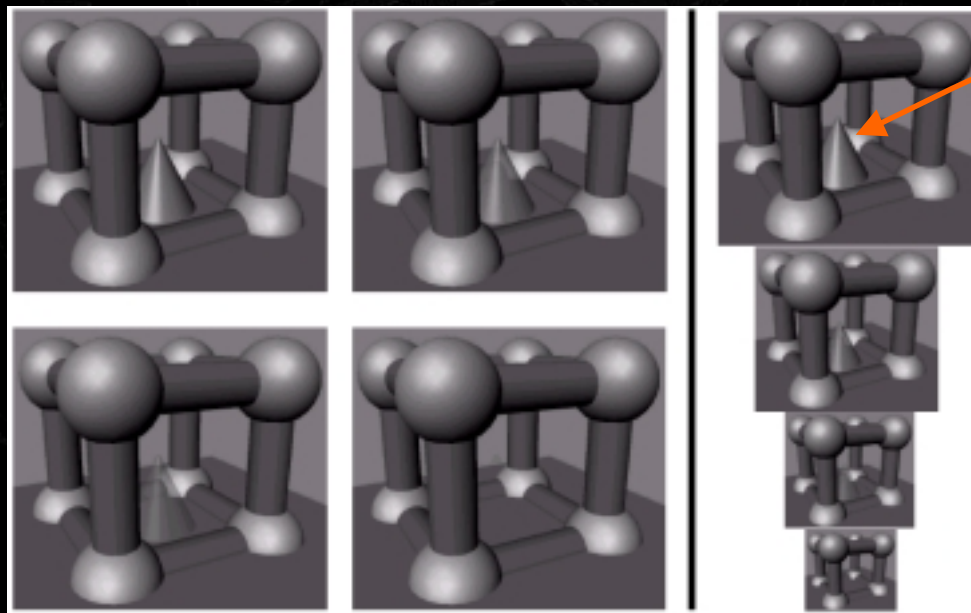
Each LOD model must be modeled separately.

Popping often occurs when switching from one model to another.

Blending between models by having overlapping ranges may be possible, but then *both* models must be rendered.

Alpha LOD

A simple idea: fade out the object as it gets beyond a certain range, until gone.



The cone
fades

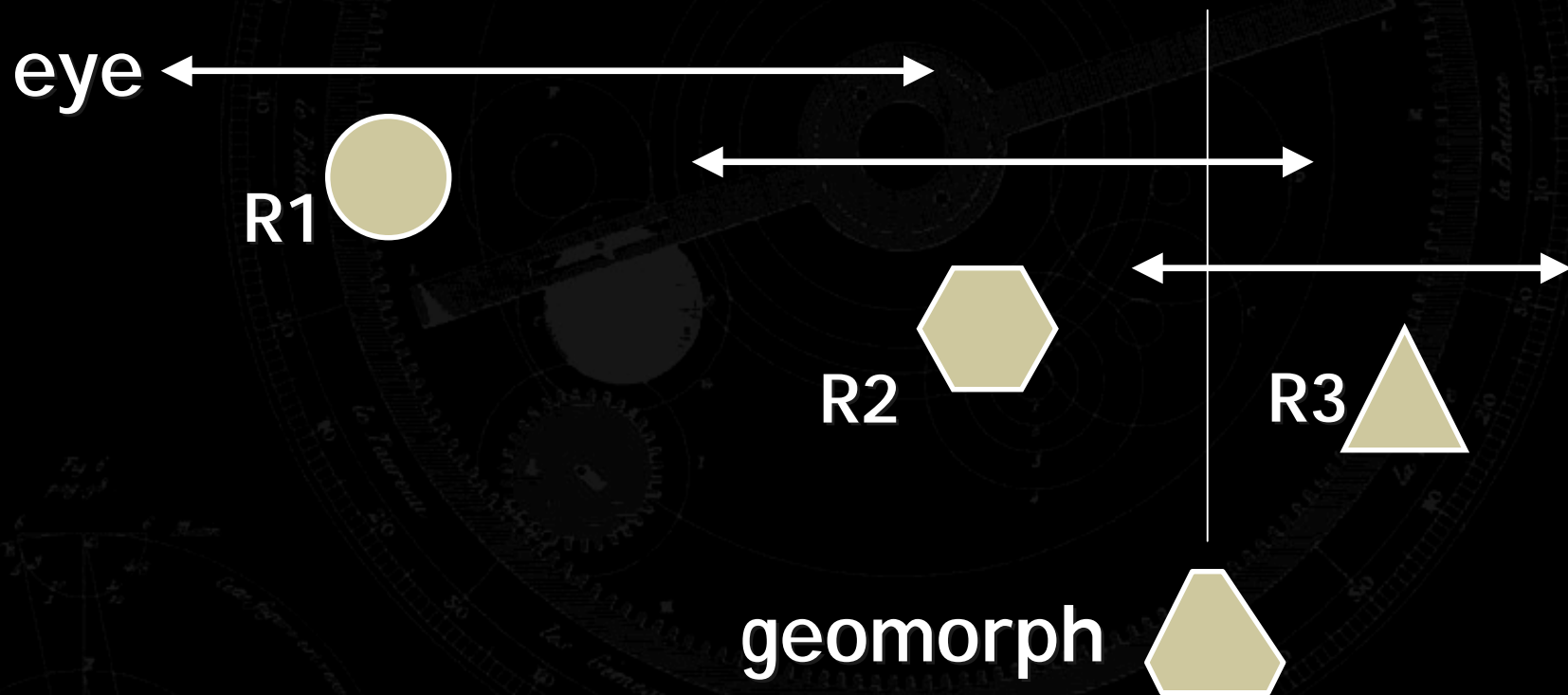
Geomorph LOD

The ideal is to smoothly transition between two different LODs.

Geomorph LODs do this by associating every vertex on the more complex model with some vertex on the simpler model.

As the blend zone is traversed, one LOD model morphs into the other.

Geomorph LODs Example



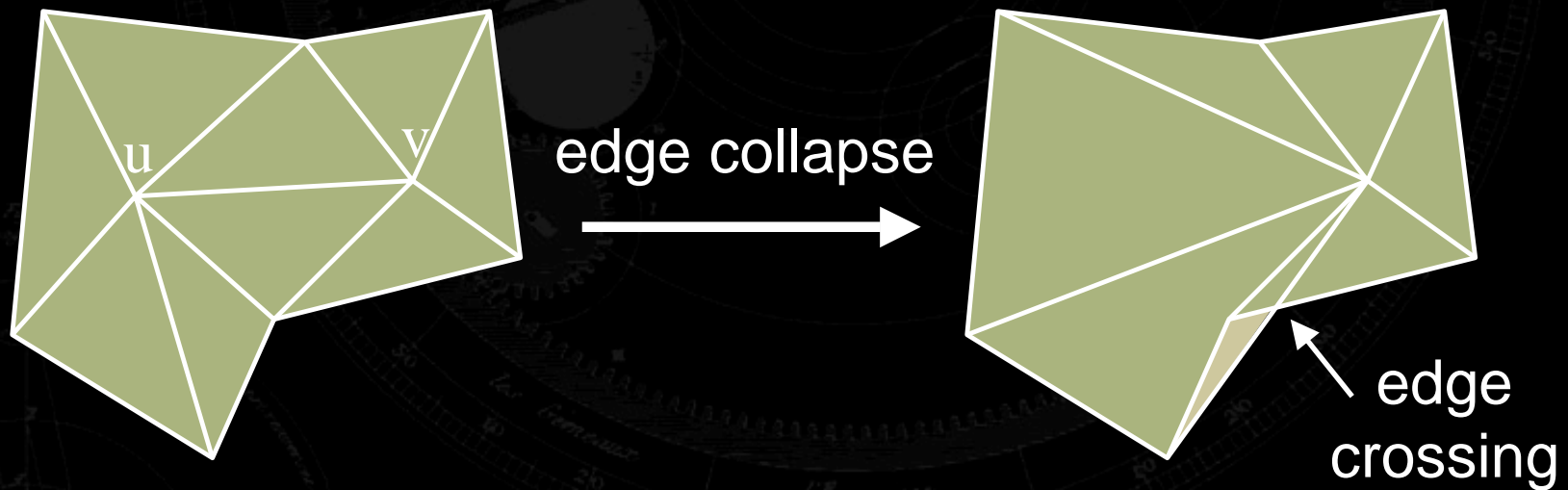
Making Geomorphs

Progressive Meshes, a.k.a. Simplification:
starting with a complex model, simplify by
removing an edge in the mesh.



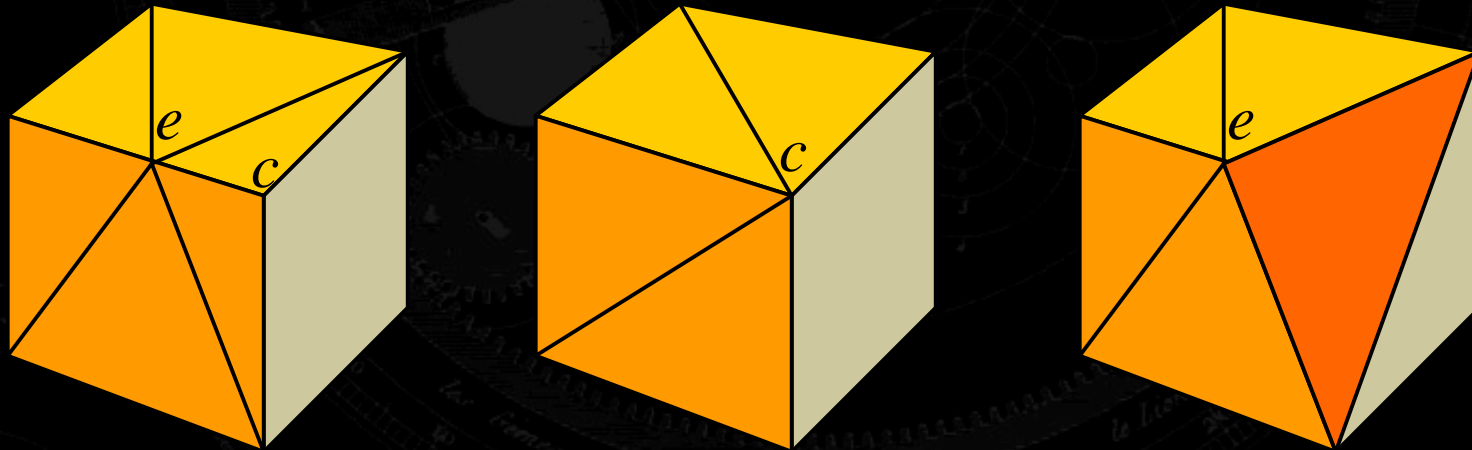
Bad Edge Collapses

Not all edges can be collapsed during simplification:



Simplification Methods

Each edge is ranked by its effect on the model.



Edge Functions

Which edge is least important is a non-trivial function, and is perception based.

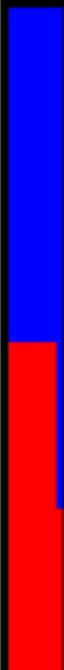
- Example: eyes and mouth more important.



Images courtesy Hugues Hoppe

Melax Demonstration

current model: cow.ply

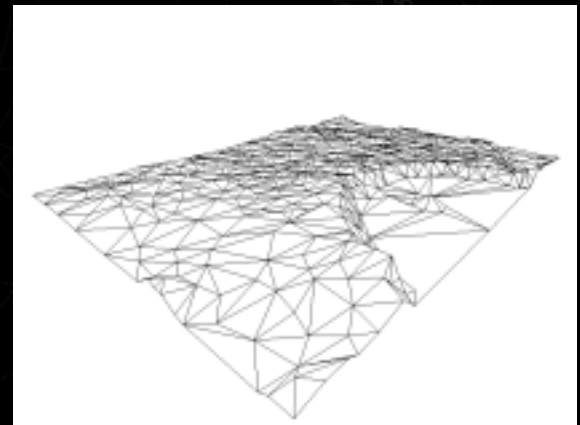
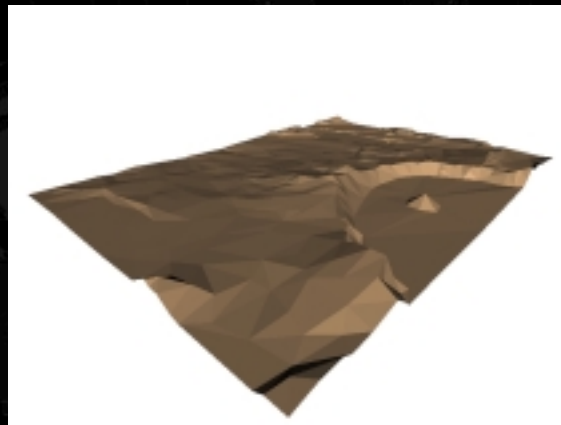
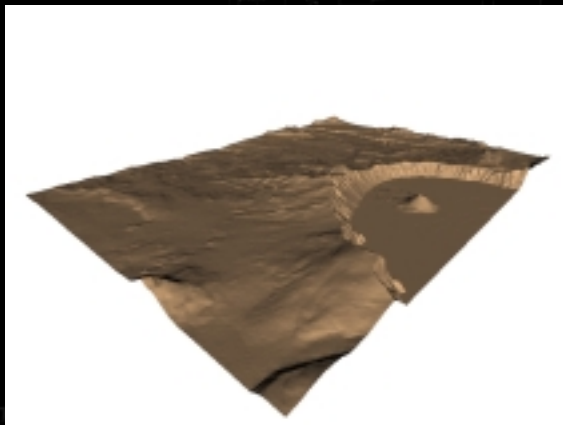


(space-bar to go to next
model, "Enter" to select)

Polys: 2900 Vertices: 1451 <-> 725 morph: 0.88
FPS: 10.92

Terrain Simplification

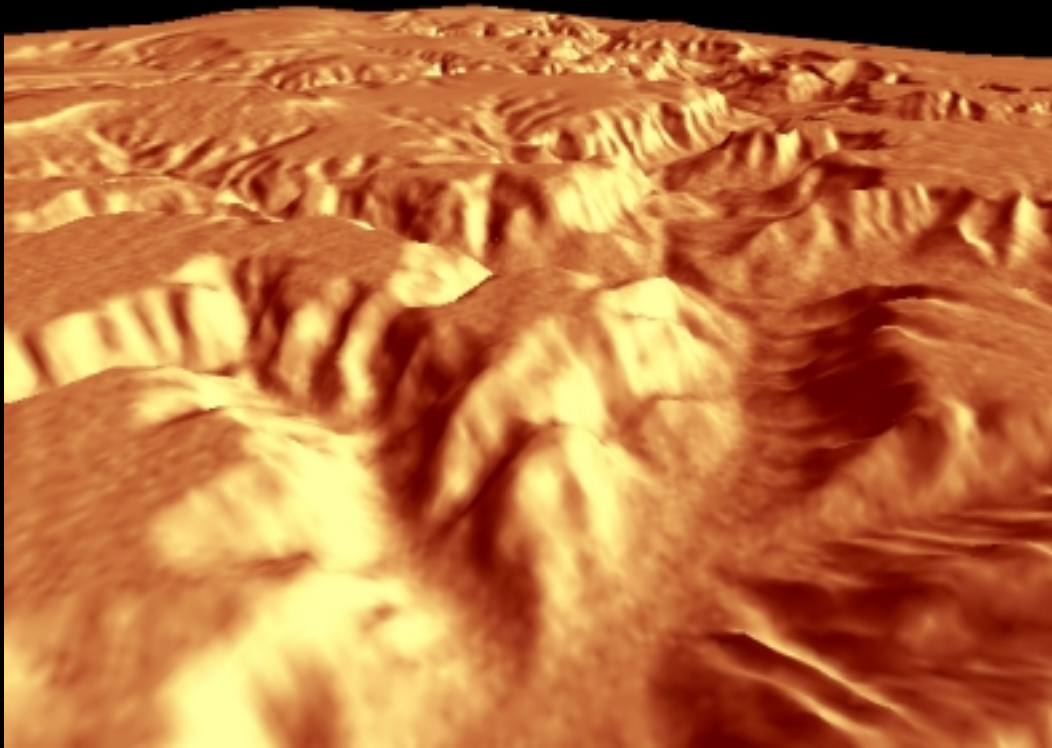
One idea is to simplify the model once:



then turn the reduced detail into texture

Images courtesy Michael Garland

Ulrich Demonstration



left mouse - rotate

right mouse - move

w - wireframe toggle

t - texture toggle

p - follow terrain

-/= - change detail

m - animated move

Screen Size Determination

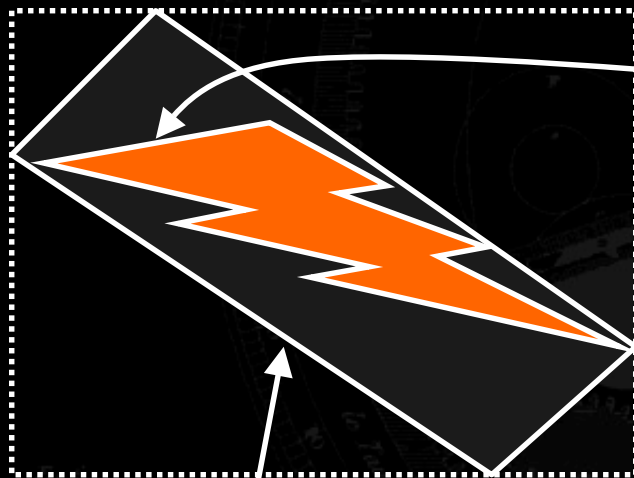
Which LOD to use is related to the number of pixels it covers. Methods:

Draw the object, count the pixels (dumb).

Use a box around the object, transform to screen, use area of rectangle.

Find area of box itself on screen
(Schamlstieg and Tobler, jgt 1999).

Size Determination Illustrated



Ideal: get area covered by object.

Typical: get 2D screen rectangle.

Compromise: get bounding box's area.

Algorithm Overview

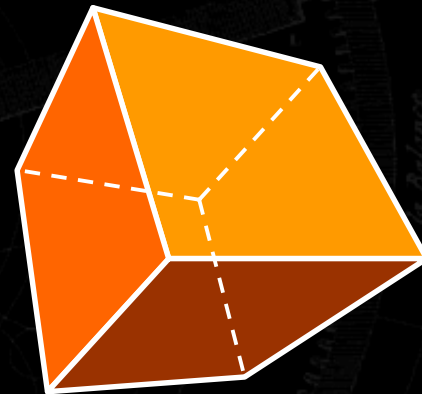
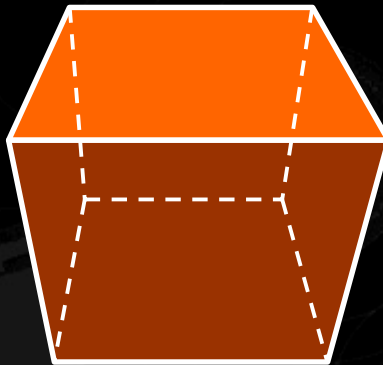
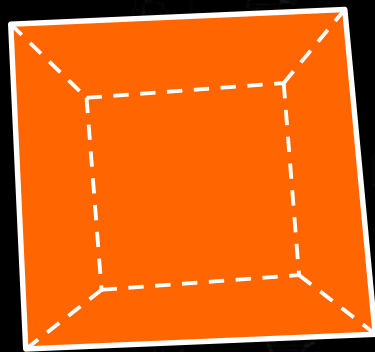
Schmalsteig and Tobler (journal of graphics tools, 1999):

Determine how many (1, 2, or 3) and which faces of the box are visible.

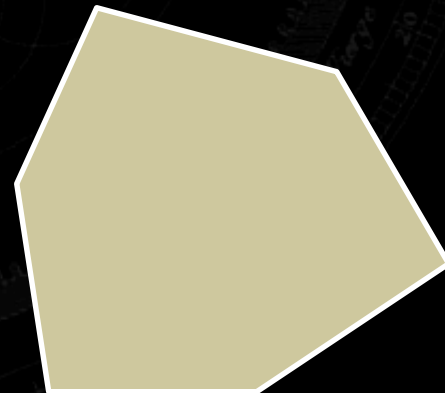
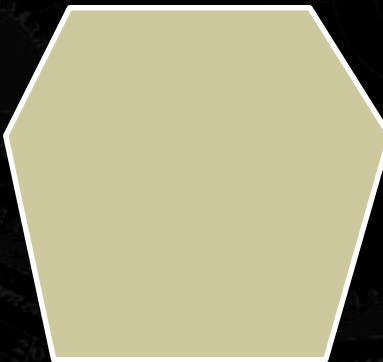
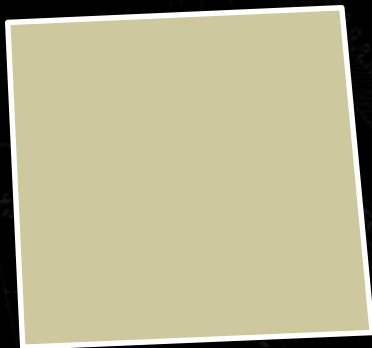
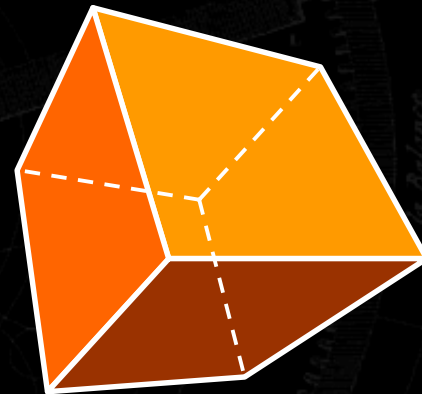
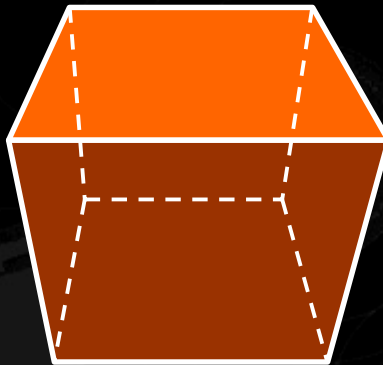
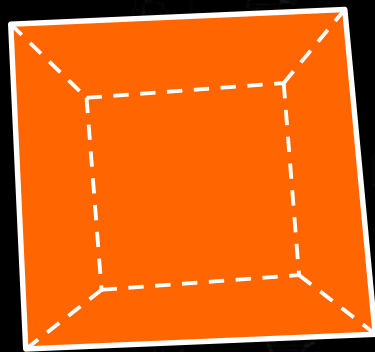
Project the vertices of the silhouette edge to the screen (4 or 6 vertices).

Compute the area from these points.

Example Boxes



Example Boxes



silhouettes

Implementation

For each pair of planes defining the box along one axis, classify the eye location as above, below, or between these planes.

$3 \times 3 \times 3 = 27$ possibilities; using bit codes for each compare gives a 43 entry table.

For each (possible) entry, give the list of the 4 or 6 silhouette edge vertices.

Compute the area: done!

More Information

The SIGGRAPH 2001 notes for this and other courses.

Book's web site

<http://www.realtimerendering.com/>

Surrender's Umbra occlusion & portal demos and manual: <http://www.hybrid.fi/umbra/>, and at Criterion's Renderware booth (look for PVS)

Code for the projected area algorithm is at <http://www.acm.org/jgt/>

Morning break



Noise Based Procedural Content and the Humble Modem

Kim Pallister
Technical Marketing Engineer
Intel Corporation
(Dean Macri, Technical Marketing Engineer, co-author)

kim.pallister@intel.com

Third-party brands and names are property of their respective owners.

Copyright © 2001 Intel Corporation. All rights reserved



Problem Statement

The problem

- Rich, complex 3D scenes have large datasets
- Internet is poor medium for distributing large datasets



The solution

- Many natural & man-made objects exhibit self-similarity - maybe 'algorithmically creatable'
- Use the processor power at the client end to generate some content procedurally.

What Is 'Noise'?

Adds 'imperfection' to algorithmic content

- makes results more natural-looking

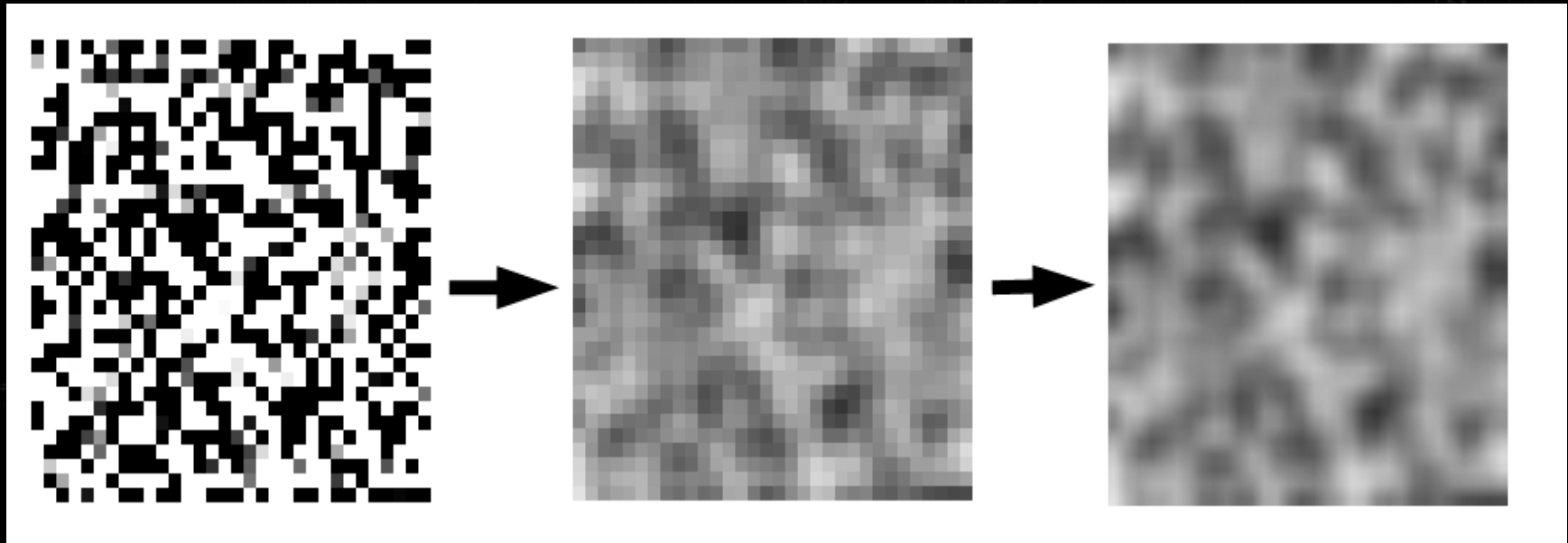
Seeded pseudo-random number generator

- Seeded because we need to be able to re-generate the same result each time

What is Perlin noise?

- Simply a smoothed version of the same random noise
- The smoothing gets rid of unnatural harsh transitions

Pre- and post- smoothed noise



Noise as function parameter

Interesting results occur when
noise is fed into functions

Some simple functions can be
offloaded to HW, others will
need to be done in SW

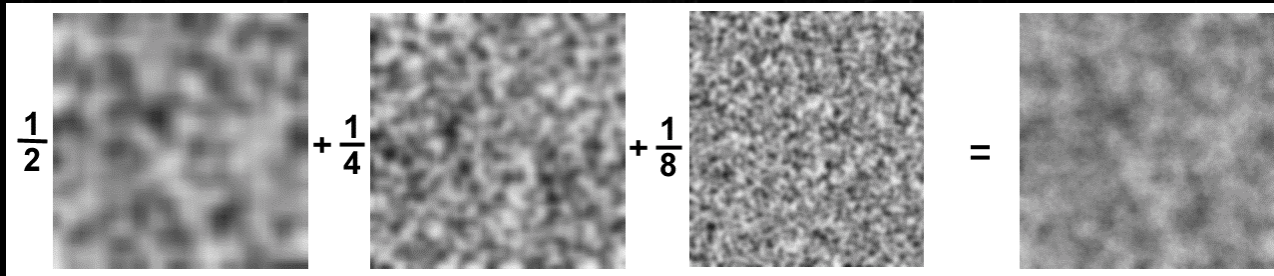
No substitutes for experimentation



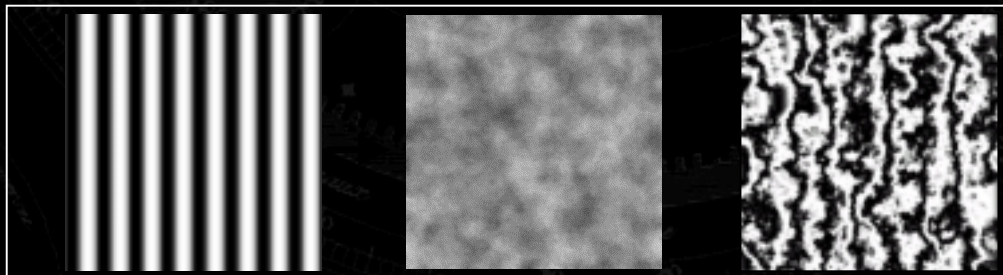
Function Examples

$1/f(\text{noise})$ (fractal sum) - clouds, fog

- Clamping or other post processing is useful too



$\text{Sin}(x + 1/f(\text{noise}))$ - marble



Application areas: Texturing

2D

- Base textures
- detail textures
- specular maps

3D

- Fog volumes

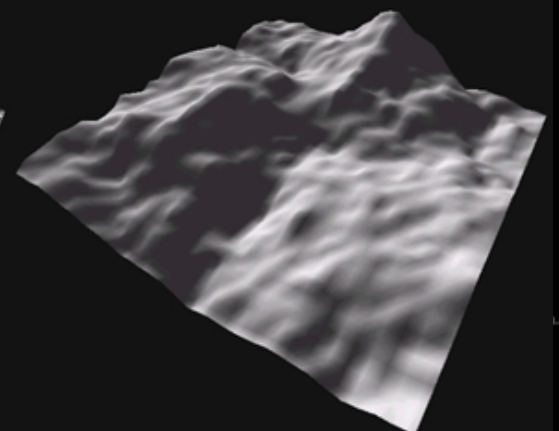
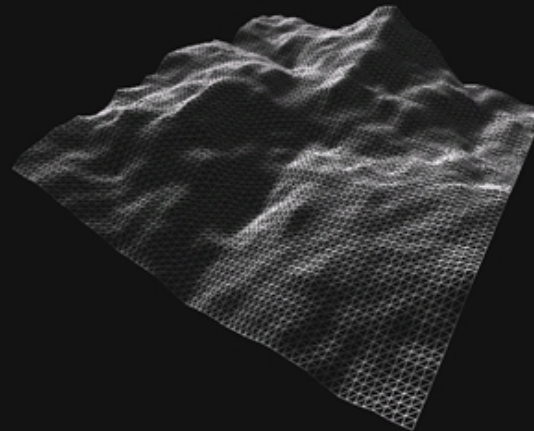
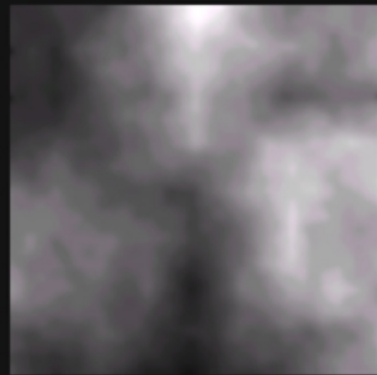
Applications areas: Modeling

Procedurally created models

- Terrain, Trees

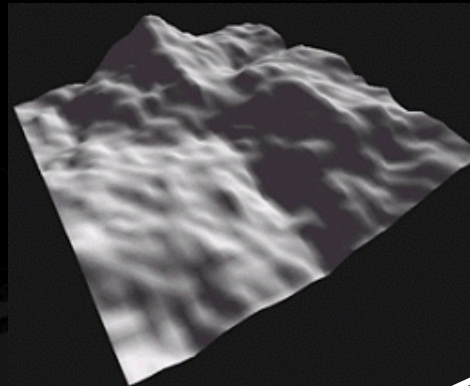
Procedurally created scenes

- distribution of trees, people in a mall, telephone poles



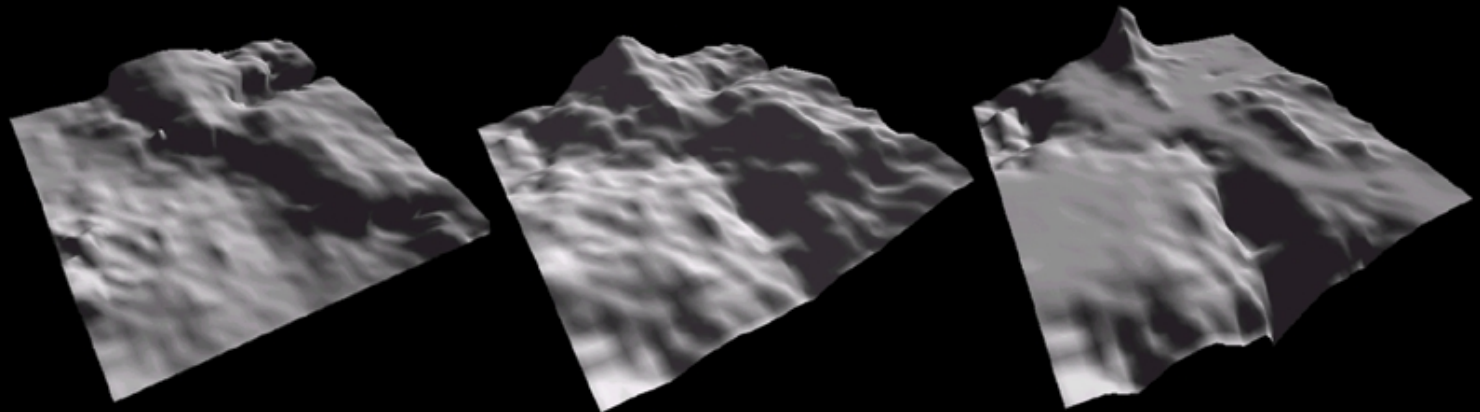
Functions + Noise = Great Results

$$\text{Height} = \text{Noise}(X, Y)$$

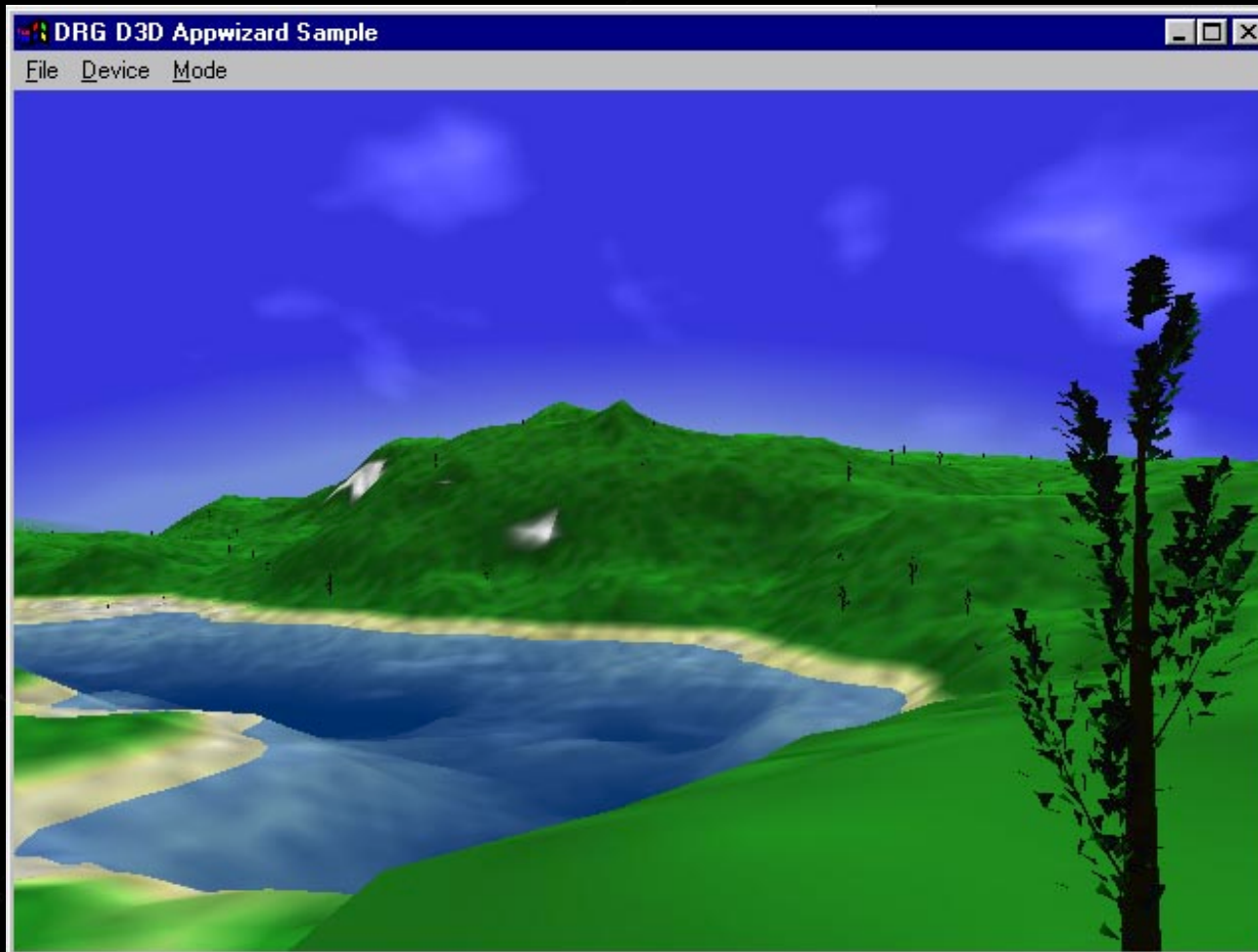


Now we can vary
these using noise!

$$\text{Height} = \text{Amplitude} * \text{Pow}(\text{Noise}(X, Y), \text{Exp})$$



Procedural content demo



Application areas: Animation and events

Great for character animation

- Head nods, shifting weight, blinking eyes

Environmental things

- When the weather changes
- How often cars drive by
- When does the cable guy show up :-)

Demo

Issues

Numerical error on different systems can have different results

- Not a big deal for textures
- Could be a big problem for procedural geometry, events, animation
- Especially in multi-user environments

Algorithm vs artist

- Results may not be what artist intended
- Need to integrate into tools



Call to action

Experiment with procedural content
creation

Don't limit it only to textures.

Character animation is a natural fit

Resources

Ken Perlin's HardcoreGDC talk and animated face demo

- <http://www.noisemachine.com>
- <http://mrl.nyu.edu/~perlin/facedemo/>

Modeling & Texturing - A Procedural Approach

- Ebert, David (ed), Morgan Kaufmann, ISBN 0-12-228730-4

Haim Barad's procedural texture article

- www.gamasutra.com/features/programming/19980501/mmxtexturing_01.htm

Intel Developer Services articles

- www.intel.com/ids/

Game Programming Gems 2

- Deloura, Mark et al, Charles River Media, ISBN 1-58450-054-0

3D API Usage and Optimizing for Variable Platforms

Kim Pallister
Technical Marketing Engineer
Intel Corporation

kim.pallister@intel.com

Third-party brands and names are property
of their respective owners.

SIGGRAPH
2001 EXPLORE INTERACTION
AND DIGITAL IMAGES

Agenda

Why multi-platform optimization

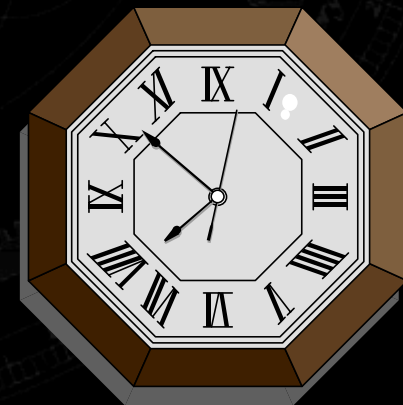
Common platform performance issues

Design guidelines

API usage

Profiling

Call to action



Why Variable Platform Optimization?

Software increasingly needs to be multi-platform

- Multi-platform games, browser plug-ins, etc

Some platforms are variable in nature

- Especially true for PC: CPU horsepower, ISA, memory, graphics subsystem

Need to have applications that run well across the board and/or adapt to target

Performance and the API

High performance used to mean hand coding top to bottom, but...

Multi-target apps foster API dependency

API provides baseline functionality

- Some of that functionality may be emulated

API may have multiple methods for accepting data & commands

- Not all of which are created equal!

So the API will abstract the target?

Not quite...

Platform architectures may vary so much
that app architecture & API usage change

Examples:

- PC application vs PlayStation2* application
- Shader/Materials on PC platform

The API can only do so much...

*Third-party brands and names are property of their respective owners.

Sub-optimal performance causes

Assuming the general from the specific

- (Works on the programmers machine, let's ship it)

Poor Design

Poorly understood system architecture (design)

API Issues:

- Start-up code done per frame
- Functionality misunderstanding
- Broken concurrency
- Poor data flow/data copies, Lack of batching
- Excessive state changes

Coding for the coder, not the HW/API



Design Guidelines

Know your platform architecture and variability

- Make high level design decisions based on this
- How much can the API abstract? Where are different code paths needed?

Know your API's functionality

- What's fast? What's slow? What works well on each platform and how does it vary across platforms?

Know your limits

API Issues: Don't do it if you don't need to

Start-up code should be done...

- (Hint: not per frame)

Creation of resources should be done ahead of time

- Vertex Buffers/Index Buffers
- Texture Surfaces
- Vertex/Pixel Shaders
- State Blocks

API Issues: Functionality

Poorly understood API functionality

- What are the optimal functions? How do/could they work?
- What path does the data take?
- E.g. How much of a VB transformed on HW vs SW?

Assumed API functionality

Lack of trust in API functionality

Untested platforms

API Usage Issues: Concurrency

Graphics subsystem operates in parallel with host CPU

- As long as one isn't waiting for the other

Typical causes for loss of concurrency

- Trying to read data back from graphics device
- Accessing shared resources at the wrong time

API Usage Issues: Concurrency

Avoid touching vidmem bits (locking)

- Forces CPU to idle while command queue flushed

Hand off vertex data in large chunks

- definitely over 100 verts unless unavoidable
- diminishing returns over 2000 vertices

Beware API calls that can stop concurrency

- e.g. GetDC in DirectDraw will issue a lock
- (GDI is always a no-no)

API usage issues: Data management

Natural assumption that app can do best job of managing vertex & texture memory

- This is increasingly false!

Sub optimal API usage will cost you

- Extra copies of data will result in performance & memory overhead
- Data may not be in the optimal place for target system (e.g. optimized system VB's)

API usage issues: Data management

Allow the API & driver to manage memory for you

- Yes, you can find ways to outperform the API...
- ..they'll work on your system but break on others

Use cleanest semantics for data hand off

- e.g. Direct3D: Index/Vertex buffers, OpenGL: Compiled vertex arrays
 - Make sure to use the right creation flags!
- Use indexed prims, use strips where you can

API usage models: General optimization tips

Above all, maintain concurrency

Avoid data copying and mismanagement

Batch data at 100-2000 verts per call as much as possible

Avoid state changes, sort by state change

Avoid redundant state changes

Never Get() anything

- View HW as one-way

Use Clear(), not tri's

State changes costs (most to least costly)

VS/PS change

VB/FVF/VA change

Texture change

Render state setup

SetRenderTarget (Direct3D only)

New matrices

Fog table states

Update light states

Turning lights on/off

**Share
Where
Possible**

SW vs HW

Some functionality may be found in HW on some systems, SW in others

- Some things in SW may still be fast (e.g. Vertex Shaders under DirectX8)

Trying to theorize about HW performance based on SW assumptions will rarely work

- e.g. Vert & Texel caches throw off BW calculations
- e.g. Texture swizzling
- No substitute for testing (IHVs can help here!)

Use API & HW friendly methods

- E.g. DrawPrimVB vs DrawPrimUP

*Third-party brands and names are property of their respective owners.

Tools for API Performance Analysis

Roll your own

- Offers the best control over profiling, potential for low overhead
- Lots of work, could introduce error

Off the shelf

- Several exist
- Minimal overhead added by profiler
- Robust UI and analysis tools mean testing can be done by non-programmers

Limits of Profilers

Where time is spent isn't necessarily where the bottleneck is

- Runtime could be spinning while waiting for HW

Understand difference between
fillrate/throughput/CPU limited

- May need to modify your tests to determine cause

Is the bottleneck in the runtime or the way that you use it?

LAB Exercise

Profiling API Usage of a Sample 3D application



Call to Action



Have faith in the API & drivers

- ...But don't trust them

Test often! (Not only 2wks before shipping!)

- Compare to past results throughout

Understand what's under the hood

- Understanding API data & program flow allows spotting of potential problems
- Helps you design correctly
- Use the tools, talk to the vendors

Profile as many permutations as possible

- No substitute for profiling on the target system
- Leverage available tools

Resources

Microsoft Direct3D FAQ

- msdn.microsoft.com/library/techart/dxfaq.htm

OpenGL performance notes

- www.opengl.org

Real Time Rendering

- Moller, Haines - ISBN 1-56881-101-2

Intel Developer Services

- <http://www.intel.com/ids/>

Resources for Profiling Tools

Intel Vtune

- <http://developer.intel.com/vtune/>

3D Pipeline GLAnalyze

- Supports OpenGL applications
- <http://www.3dpipeline/products/gl/glanalyze.htm>

Nvidia stats driver (registered developers only)

- <http://www.nvidia.com/developer.nsf>

Intellgraphics Intellibench

- Supports DirectX (older versions only)
- <http://www.intellgraphics.com/ibench.htm>

Bandwidth reduction: Compact culling and Bezier tessellation

Haim Barad
Staff Engineer
Intel Corporation

barad@acm.org

Compact Backface Culling

Highlights

- Reduce the storage/bandwidth required for facet normals by 50%
- Perform culling at front-end in object space
- Optimized for SIMD computation

Efficient Culling

Store facet normals to avoid calculations on the fly

- Sign check of dot product with viewing vector
- Require positions of polygon and view as well as facet normal

$$(P - V) \bullet N$$

Storage requirements

```
struct {  
    float x,y,z;  
} Vector;
```

```
struct {  
    Vector position; // position on polygon - 12 bytes  
    Vector normal;   // normal of polygon - 12 bytes  
    WORD p1, p2, p3; //indices to vertex pool - 6 bytes  
    WORD stub;       // needed to avoid 2 byte alignment  
                    // stalls and filling a cache line.  
} faceData; // total size 32 bytes
```

Phase 1: Object space culling

Front end culling is faster

- 10-20% frame rate boost on many apps
- Eliminate polygons BEFORE transform

Transform viewer into object space

- Facet normal is precalculated
- Position of normal is one of the vertices

Rewrite culling test

Culling test can be rewritten:

$$(P - V) \cdot N$$

Precalculate!!!

Don't need P


$$(P \cdot N) - (V \cdot N)$$

Compact Representation

Store normal in fixed point

- Normal is “normalized” between -1 and +1.
- Represent normal in range from -32767 to +32768
- Only 2-byte representation instead of 4

Compact Storage Requirements

```
struct {  
    float pn; // precalculated position*normal  
              // 4 bytes  
    signed short normal[3]; // normal of polygon  
                          // 6 bytes  
    WORD p1, p2, p3; // indices to vertex pool  
                    // 6 bytes  
} faceData; // total size 16 bytes
```

Only half the storage requirements!

Modified Algorithm

```
for ( each face in the array){  
    // Expand the normal back to float from its fixed-point  
    // Calculate face->tn = (face->normal * Vo)  
    // Previously we calculated (P-V)*N, but we have tn = P*N  
    // (precalculated) we now need to calculate (tn - V*N)  
    if ( result <= 0){ // polygon is front-facing  
        // face->p1, face->p2 and face->p3 are indices to the  
        // vertex pool. Use them to mark the appropriate  
        // vertices "visible"  
    }  
}
```


Performance Advantages

Front-end culling

Sequential memory accesses

- Data structures can be prefetched

Data structure for culling is half the size

Sample code available on Internet at
Gamasutra website (gamasutra.com)

Performance Results

Ratios of

- A = compact scalar/non-compact scalar
- B = compact SIMD/non-compact SIMD
- C = compact SIMD+prefetch/non-compact SIMD

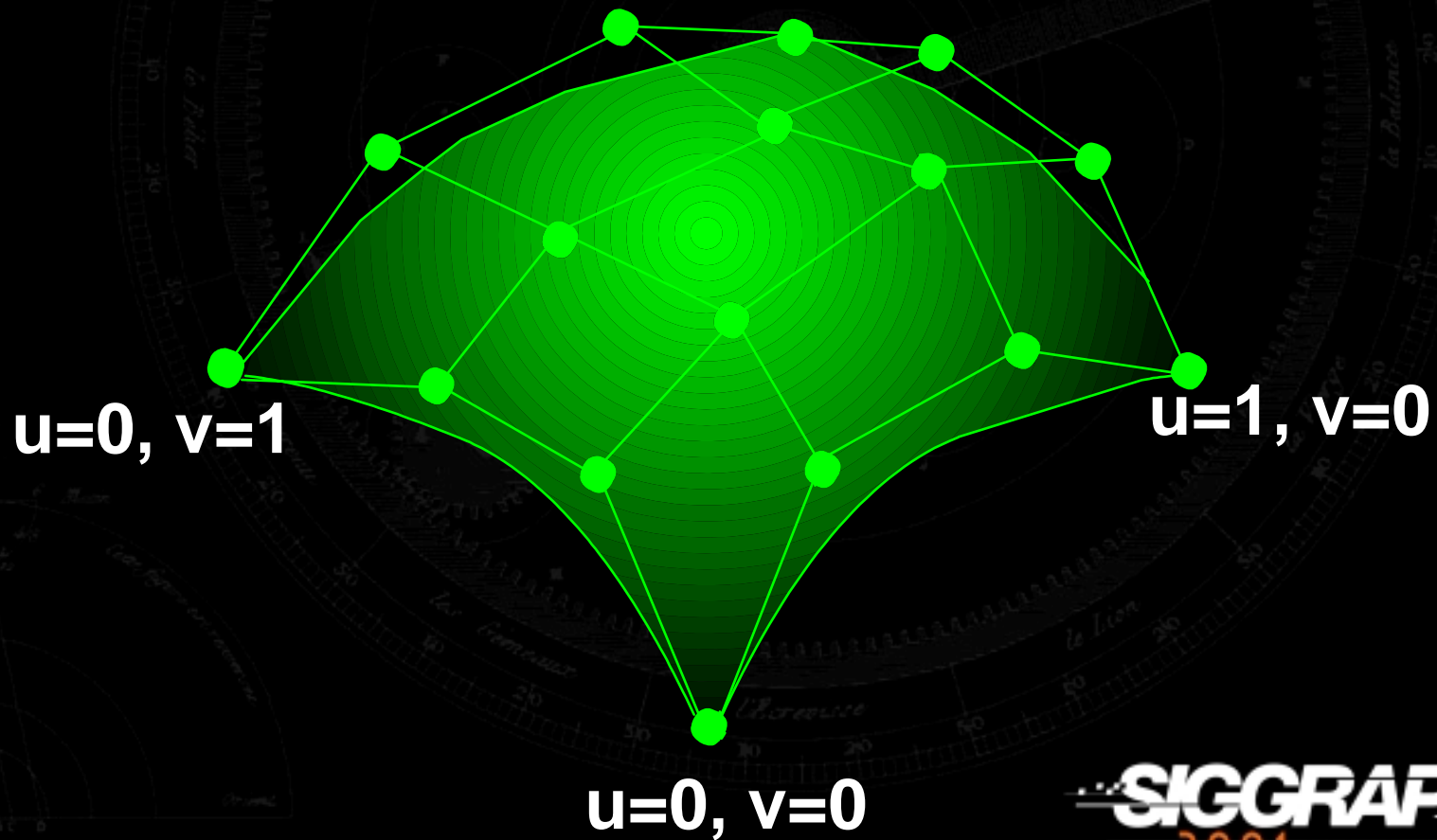
Spherical Object			
# tris	400	6.4K	20K
A	1.15	1.2	1.2
B	1.45	1.6	1.7
C	2.3	2.7	2.7

Bezier Surface Tessellation

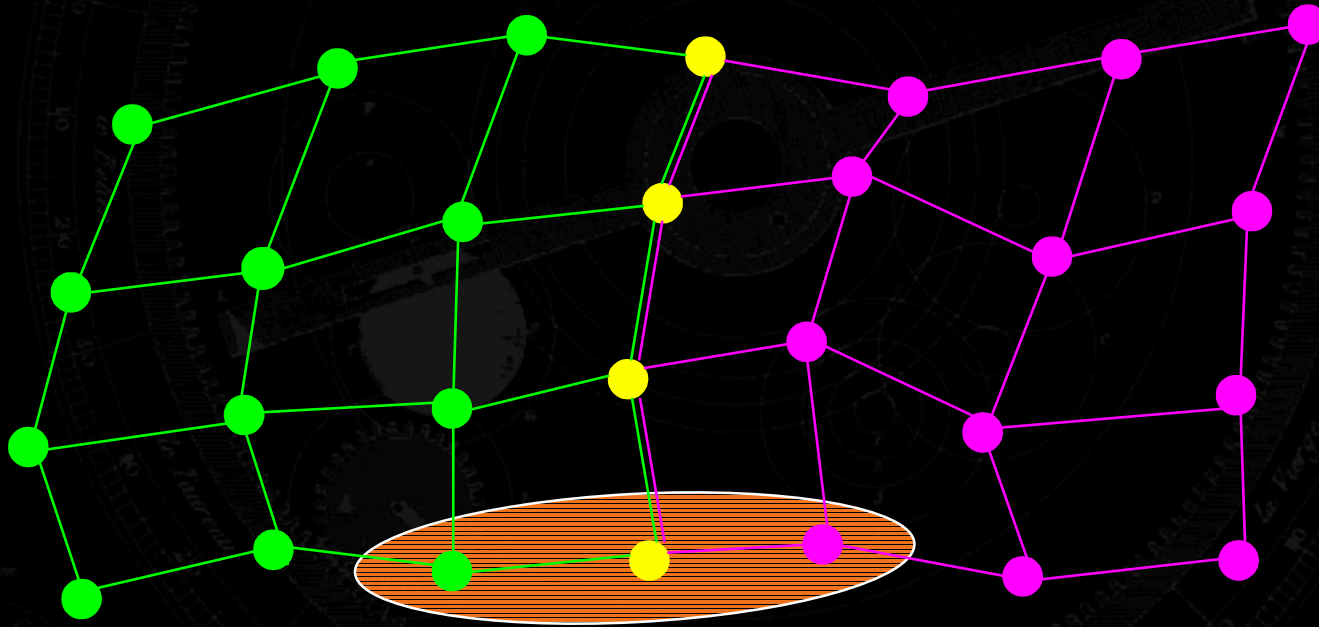
Growing importance because of Internet
We'll concentrate on 4x4 patches
Fits 4-wide SIMD computation

$$S(s, t) = \sum_{i=0}^3 \sum_{j=0}^3 \mathbf{P}_{i,j} B_{i,n}(s) B_{j,m}(t)$$

Bezier patch: 4x4 grid



Preserve G1 Continuity



**Collinear control points
along common edge**

Pipeline configurations

SW T&L

Transform control points

Tessellation

Per vertex lighting

Rasterization

HW T&L

Tessellation

Transform vertices

Per vertex lighting

Rasterization

A little math...

Surface

$$S(s, t) = \sum_{i=0}^3 \sum_{j=0}^3 \mathbf{P}_{i,j} B_{i,n}(s) B_{j,m}(t)$$

Normals

$$\begin{aligned} S_{Normal}(s, t) &= \frac{\partial S(s, t)}{\partial s} \times \frac{\partial S(s, t)}{\partial t} \\ &= \sum_{i=0}^n \sum_{j=0}^m \mathbf{P}_{i,j} B'_{i,n}(s) B_{j,m}(t) \times \sum_{i=0}^n \sum_{j=0}^m \mathbf{P}_{i,j} B_{i,n}(s) B'_{j,m}(t) \end{aligned}$$

And the basis functions...

$$B_{0,3}(u) = (1-u)^3$$

$$B_{1,3}(u) = 3u(1-u)^2$$

$$B_{2,3}(u) = 3u^2(1-u)$$

$$B_{3,3}(u) = u^3$$

$$B'_{0,3}(u) = -3(1-u)^2$$

$$B'_{1,3}(u) = 3(1-u)^2 - 6u(1-u)$$

$$B'_{2,3}(u) = 6u(1-u) - 3u^2$$

$$B'_{3,3}(u) = 3u^2$$

Tessellation: 3 basic steps

1. Take samples of the parametric surface
 - We use uniform sampling
 - These techniques work for any method
2. Connect samples into triangles
 - Generate indices list
3. Generate tessellated surface vertices
 - Generate vertex structures
 - Generate in screen space – save transform work!
 - Valid for Rational Bezier surfaces

Data Structures & Classes

CBezierTessellation Class

- Holds indices data for current tessellation level
- Holds precalculated values for B and B'

CBezierSurface & CBezierPatch Classes

- Connects patches to form complex surfaces

SIMD implementation

Evaluate surface position & normal for four sample points simultaneously

For improved cache locality, use

$$B_{0,3}(s_0), B_{0,3}(s_1), B_{0,3}(s_2), B_{0,3}(s_3),$$

$$B_{1,3}(s_0), B_{1,3}(s_1), B_{1,3}(s_2), B_{1,3}(s_3),$$

$$B_{2,3}(s_0), B_{2,3}(s_1), B_{2,3}(s_2), B_{2,3}(s_3),$$

$$B_{3,3}(s_0), B_{3,3}(s_1), B_{3,3}(s_2), B_{3,3}(s_3),$$

Process 4 at a time!

SIMD implementation (cont.)

Use 32 FP numbers (four cache lines)

- 4 s values x 4 basis + 4 t values x 4 basis
- Use prefetch to ensure no cache misses

Control points are expanded four times

- Generates four vertices in parallel

Position calculation

Convert to rational Bezier surfaces

- Persistent in projective transformations

$$S(s, t) = \frac{\sum_{i=0}^n \sum_{j=0}^m P_{i,j} B_{i,n}(s) B_{j,m}(t)}{\sum_{i=0}^n \sum_{j=0}^m W_{i,j} B_{i,n}(s) B_{j,m}(t)}$$



Code details...

Sources available in supplemental notes &
Gamasutra website (gamasutra.com)

Fully coded in C++ using SIMD classes

Performance results

Bezier teapot object

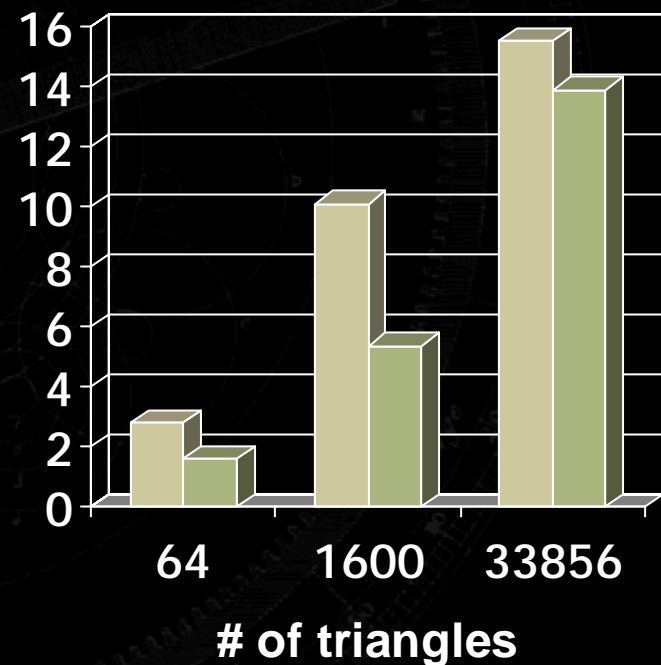
- 306 control points
- 32 patches

Tessellate position

One directional light

Pentium® III processor @
500 MHz

Mclks



Legacy SIMD

LUNCH



SIGGRAPH
2001 EXPLORE INTERACTION
AND DIGITAL IMAGES

Power Programming with Streaming SIMD Extensions 2 Labs

Alex.Klimovitski@intel.com
Tools & Technologies Europe

Now We Will:

- Explore the usage modes for the Streaming SIMD Extensions 2 (SSE2)
- Jump-start using SSE / SSE2
- Port from x87 to SSE FP
- Enhance SSE FP code with SSE2 Integer
- Use SSE/SSE2 Intrinsics and Vector Classes

Our Agenda

Port x87-intensive code to SSE FP

Prepare data for SSE with SSE

[De]Compress data with SSE2

Summary

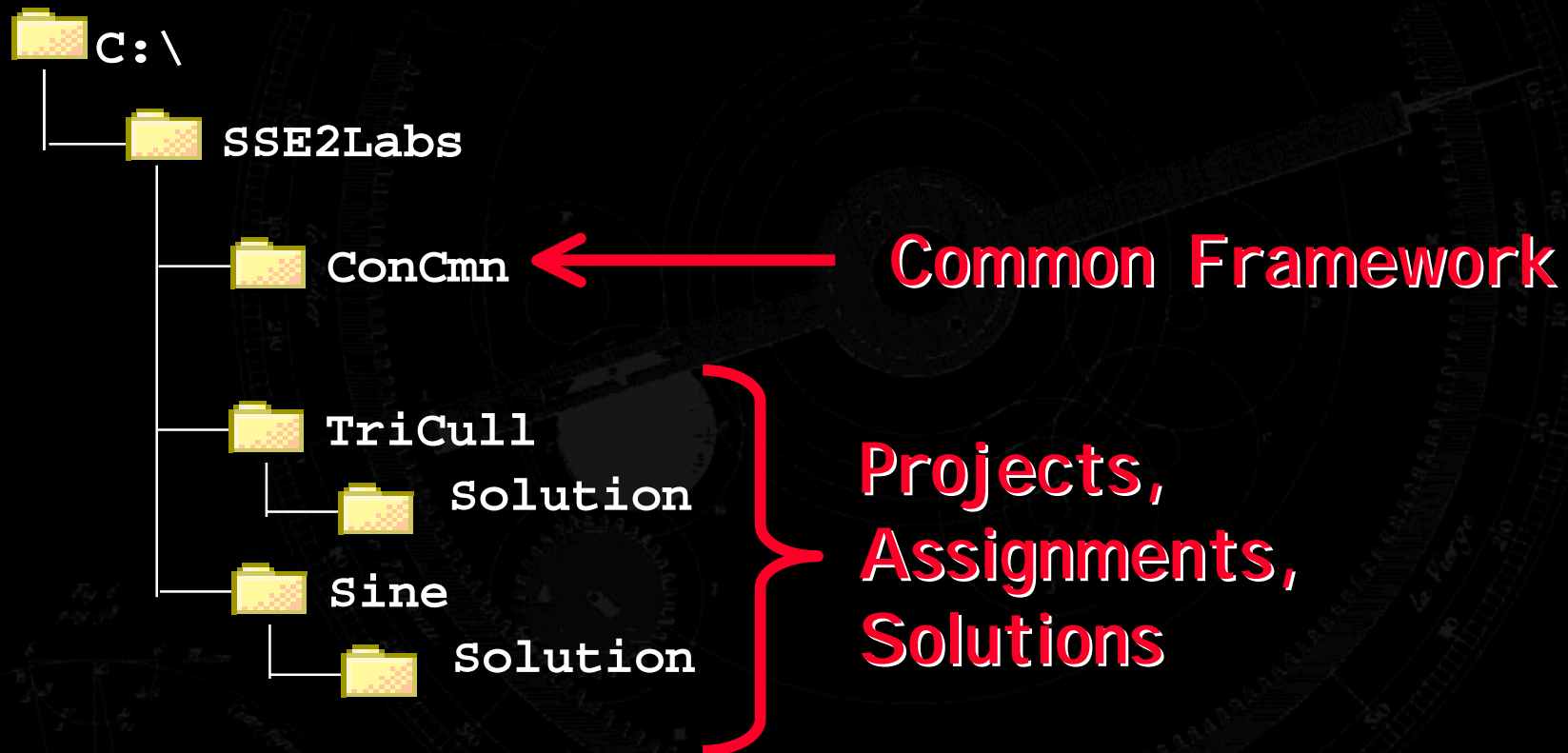
Our Tools

Microsoft* Visual C++* 6.0
Intel Compiler

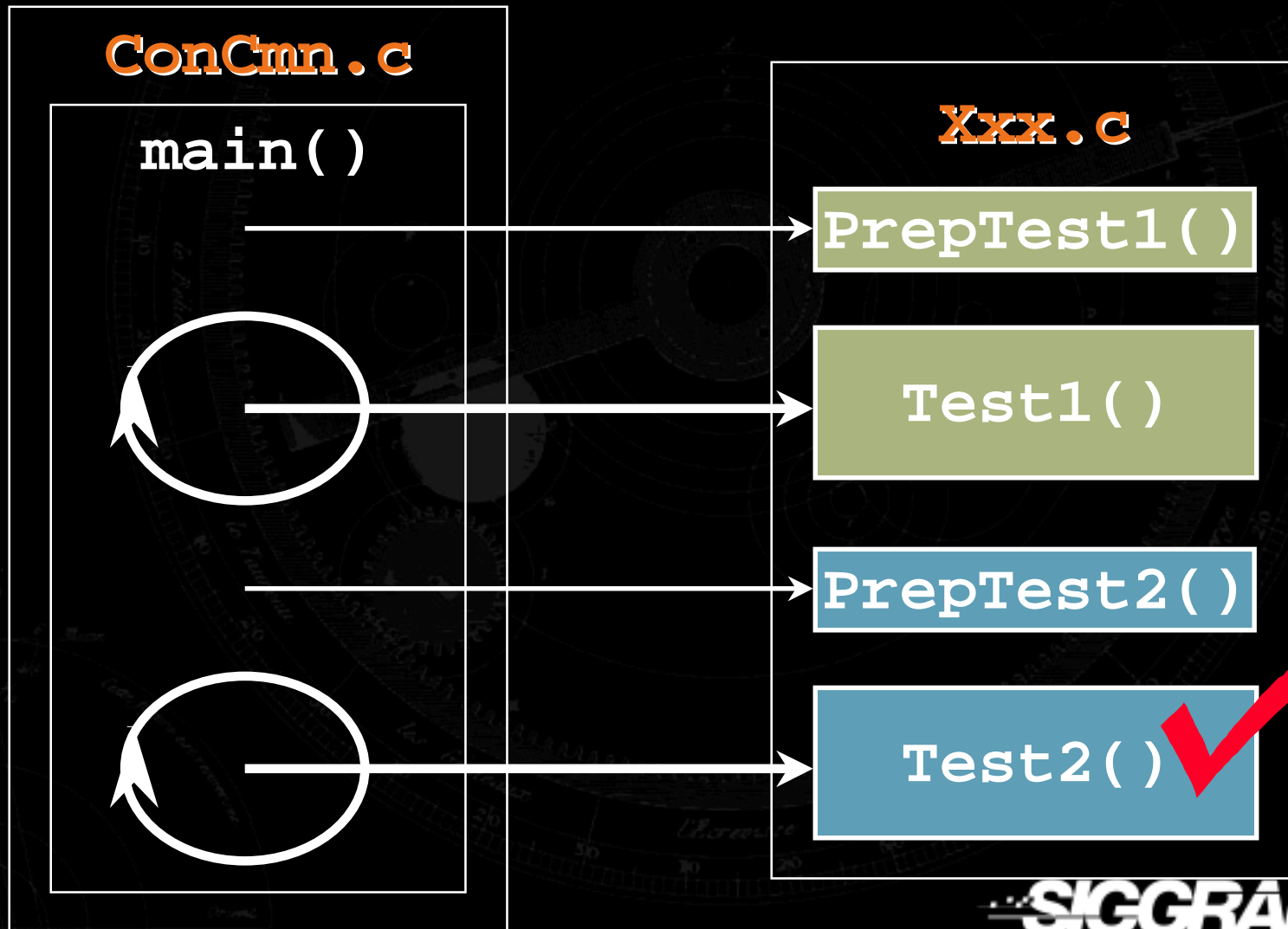
SIGGRAPH
2001
EXPLORE INTERACTION
AND DIGITAL IMAGES

*Other brands and names are the property of their respective owners.

Lab Directory Structure



Our Framework



Our Framework, Main File

ConCmn\ConCmn.c

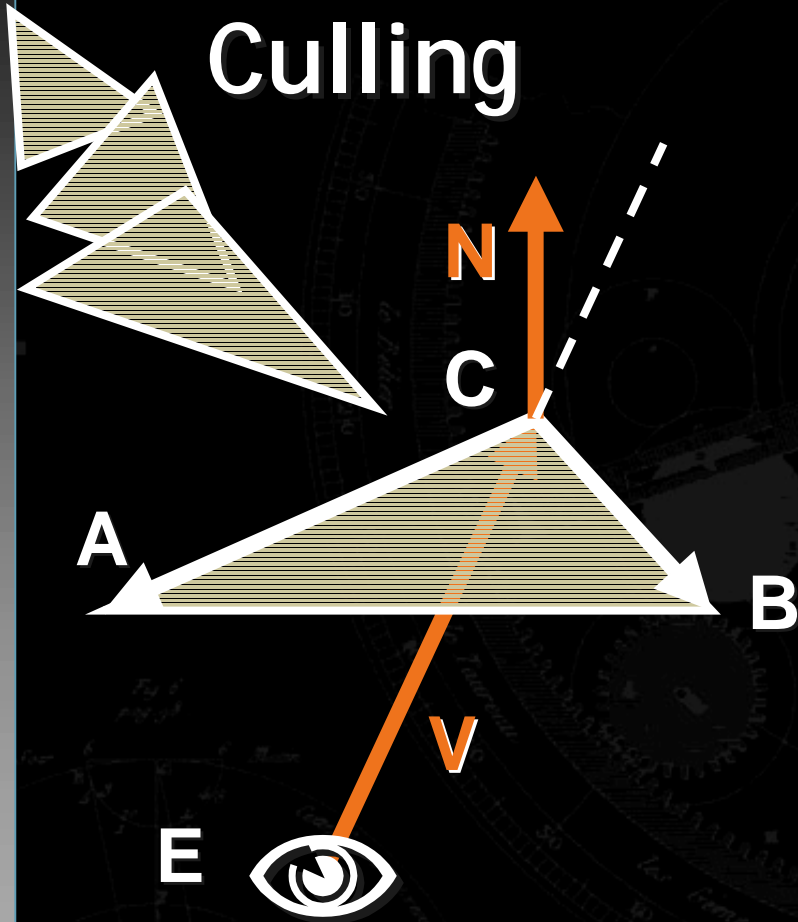
```
// Test1
PrepTest1(); // allocate & initialize memory
ReadTscSync(&cA);
for (i = 0; i < g_nTestCount; i++)
    Test1(); // do the test
ReadTscSync(&cB);
c1 = cB - cA;

// similar for Test2
PrepTest2(); // allocate & initialize memory
ReadTscSync(&cA);
for (i = 0; i < g_nTestCount; i++)
    Test2(); // do the test
ReadTscSync(&cB);
c2 = cB - cA;

Finalize(); // check the results, free memory

// compare the time for Test2 vs. Test1
print( (c2 - c0) / (c1 - c0) * 100.0 );
```

Facet Normal-Based Triangle Culling



Needed per triangle:
 $C \cdot N$ -- scalar
 N -- vector (3 coord)

Facet Normal:

$$N = (A - C) \times (B - C)$$

View Vector:

$$V = C - E$$

Cull Test:

$$t = \boxed{V} \cdot N$$

if $t \leq 0$ then PASSED

*Do we need C
for every tri?!*

Modified Cull Test:

$$t = (C - E) \cdot N$$

$$t = C \cdot N - E \cdot N$$

Our Agenda

Port x87-intensive code to SSE FP

Prepare data for SSE with SSE

[De]Compress data with SSE2

Summary

What Is SIMD? - Single Instruction, Multiple Data

Scalar processing

- traditional mode
- one operation produces one result

X

+

Y

X + Y

X

Y

X + Y

SIMD processing

- with SSE / SSE2
- one operation produces multiple results

x3 x2 x1 x0

+

y3 y2 y1 y0

x3+y3 x2+y2 x1+y1 x0+y0

SSE / SSE2 SIMD Data Types

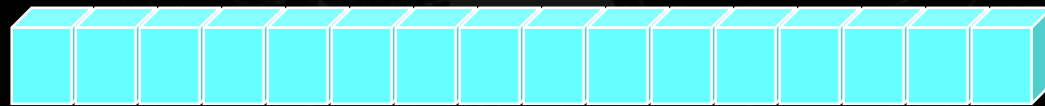
Anything that fits into 16 byte!



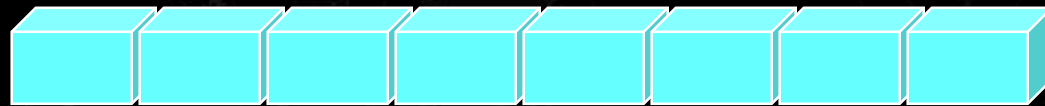
4x floats



2x doubles



16x bytes



8x words



4x dwords



2x qwords



1x dqword



Let's Get Started!

Open **TriCull\TriCull.dsw** , file **TriCull.cpp**

Consider data declaration & initialization

Test1() uses x87 **prepninfo()**, **tricull()** functions and **FNINFO** structure

Test2() uses **prepninfo_ps_A()** and **tricull_ps()** functions, **FNINFO_PS** struct



Our task: Implement SSE version **tricull_ps()!**



Step 1: Defining Data Structure for SSE Culling

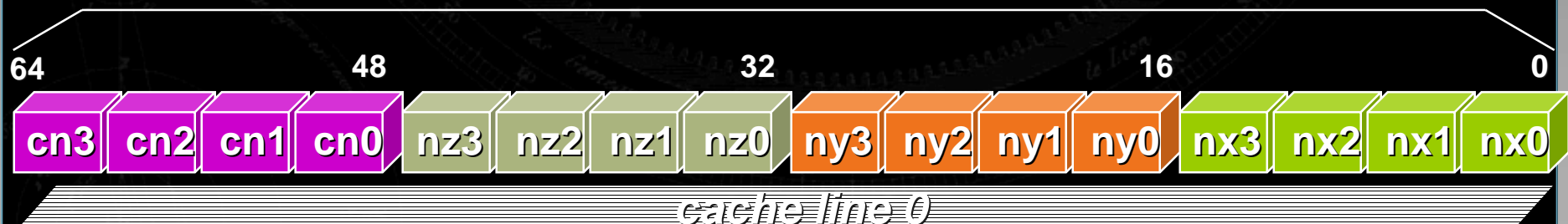
We'll be processing four triangles in parallel

Define appropriate **struct FNINFO_PS**

b,c In **prepfinfo_ps_A()**, initialize **finfo_ps** from **triinfo**

- Hint: Use $i \gg 2$ to index the structure,
 $i \& 0x3$ to index the element


finfo_ps[0]



Four elements – but just one memory stream!

SSE / SSE2 Intrinsics

SSE/SSE2 data types as C data types

<code>__m128</code>		four floats	<code>_ps</code>
<code>__m128d</code>		two doubles	<code>_pd</code>
<code>__m128i</code>		any ints in 16 bytes	<code>_epi8...epi64</code> <code>_si128</code>

● SSE/SSE2 instructions as C functions

`c = _mm_add_ps(a, b);`

standard prefix → `_mm_`

`add` → **operation**

`_ps` → **operand type suffix**

↑



Step 2: Implementing Culling Test with SSE

Follow comments starting with `// ***`

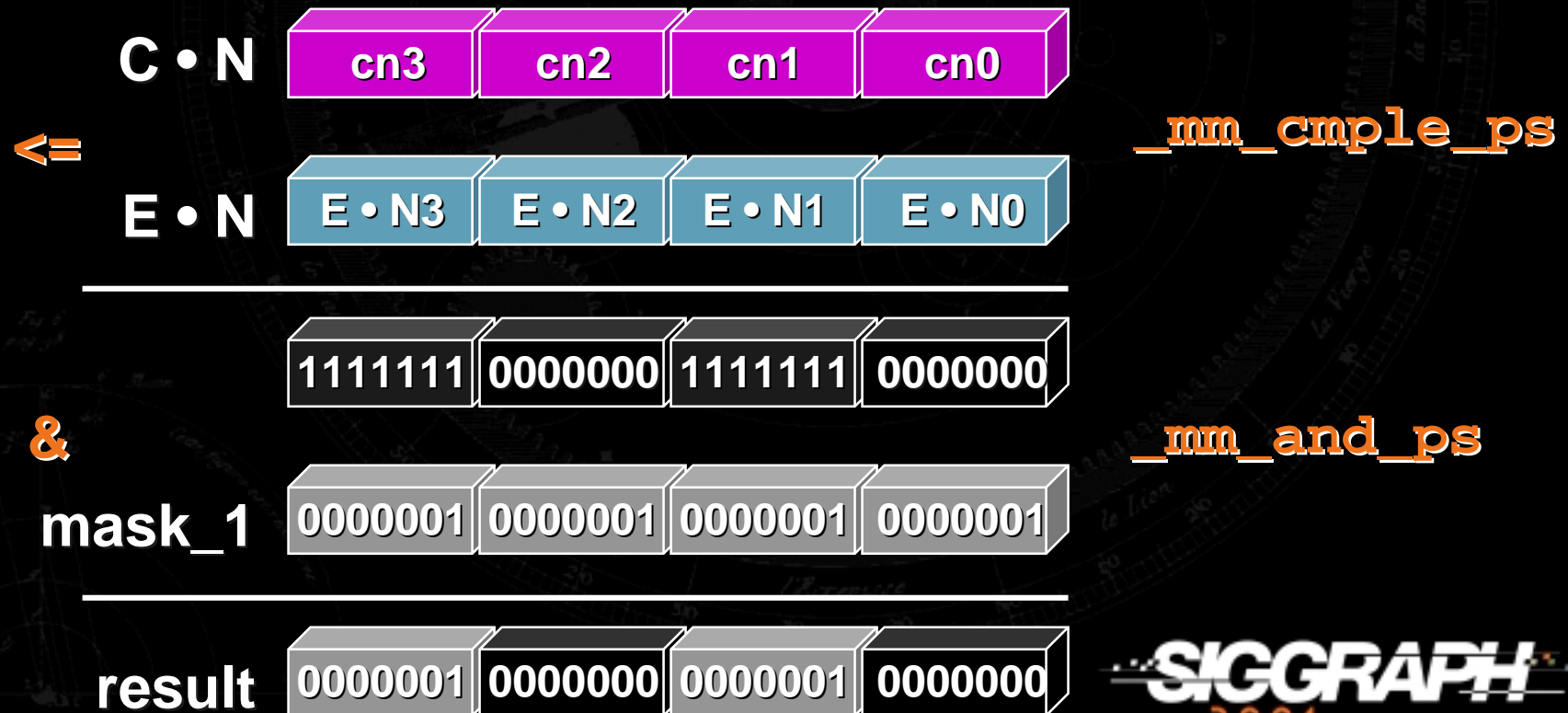
- a, b. Prepare SSE eye-vector coordinates
- c. Define the loop limit and increment
 - Hint: we are looping thru four-packed normals
- d. Load four-packed normal components
- e. Calculate four-packed dot product of the normals and the eye vector
- f. Perform the test, convert results to `BOOLs`
- g. Write out the results
- h. Try streaming results out

**Build (F7),
run (Ctrl+F5),
enjoy!**

Performing Test with SSE

We're getting SSE masks (all 1s or 0s)

The test should generate BOOLs. How?



Defining SSE Bit Masks

No valid `float` with bit pattern needed?

Define aligned static array of four integers

Load it at runtime as packed `floats`

```
#define CONST_INT32_PS(N, V3,V2,V1,V0) \  
static const _MM_ALIGN16 int _##N[] = \  
    {V0, V1, V2, V3}; /*little endian!*/ \  
const F32vec4 N = _mm_load_ps((float*)_##N);  
  
// usage example, mask for elements 3 and 1:  
CONST_INT32_PS(mask31, ~0, 0, ~0, 0);
```

Use the full power of C/C++ preprocessor!

Our Takeaway from Porting to SSE

SSE/SSE2 boost performance of FP code

Use SSE/SSE2-friendly data structure

SSE/SSE2 Intrinsics produce efficient code
without assembler

SSE/SSE2 compare & logic operations replace
branches

Our Agenda

Port x87-intensive code to SSE FP

Prepare data for SSE with SSE

[De]Compress data with SSE2

Summary

SSE/SSE2 Vector Classes




C++ wrap for a `__m128x` data types and corresponding intrinsics

Intrinsics

```
• __m128 a, b, c;  
• a = _mm_mul_ps(_mm_add_ps(a, b), c);  
•
```

Vector Classes

```
• F32vec4 a, b, c;  
• a = (a + b) * c;  
•
```

<code>__m128</code>		four floats	F32vec4
<code>__m128d</code>		two doubles	F64vec2
<code>__m128i</code>		any ints in 16 bytes	I8vec16... I64vec2

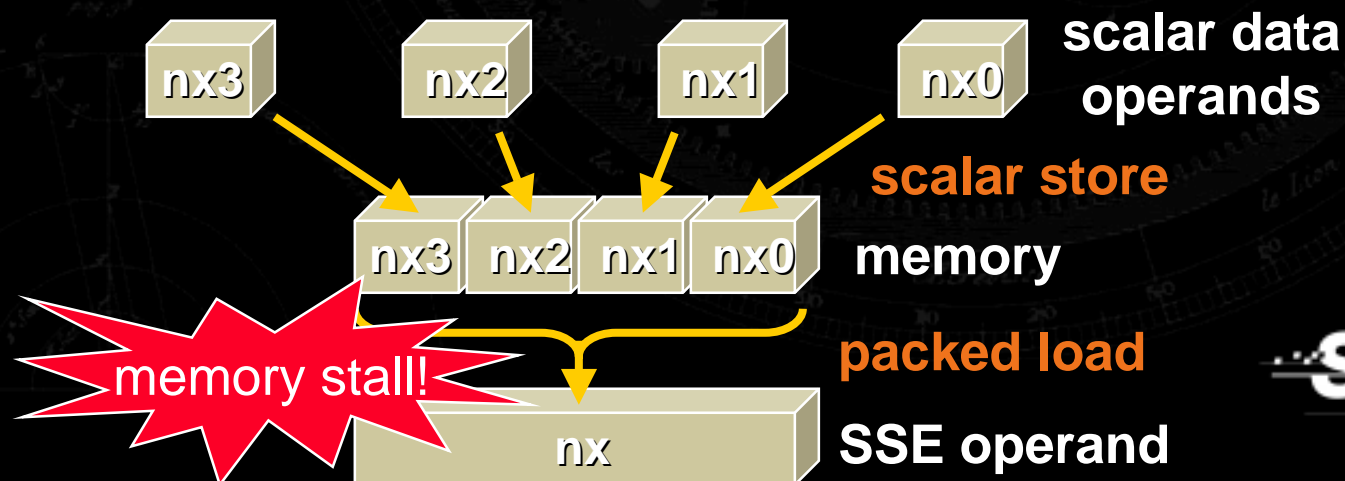


Scalar Data Preparation Problem

Move **prepfinfo()** function into **Test1()** and **prepfinfo_ps()** into **Test2()**, compare performance

Problem 1: Scalar filling of **FNINFO_PS** structures is slow

Problem 2: Memory stalls possible





Step 3: Preparing SSE Data with SSE

- Implement **vectsub()**, **dotproduct()**, **crossproduct()** using **F32vec4** Vector Class
- Assume we have **TRIINFOtoF32vec4()**. Implement **prepfninfo_ps()** using **F32vec4**
- In **PrepTest2()**, use **prepfninfo_ps_B()** instead of **prepfninfo_ps_A()**

x87 code is easy to port with C++ Vector Classes!

Now what about TRIINFOtoF32vec4() ?

Data Swizzling Problem

- The way we have it: array of structures



- The way we need it: structure of arrays

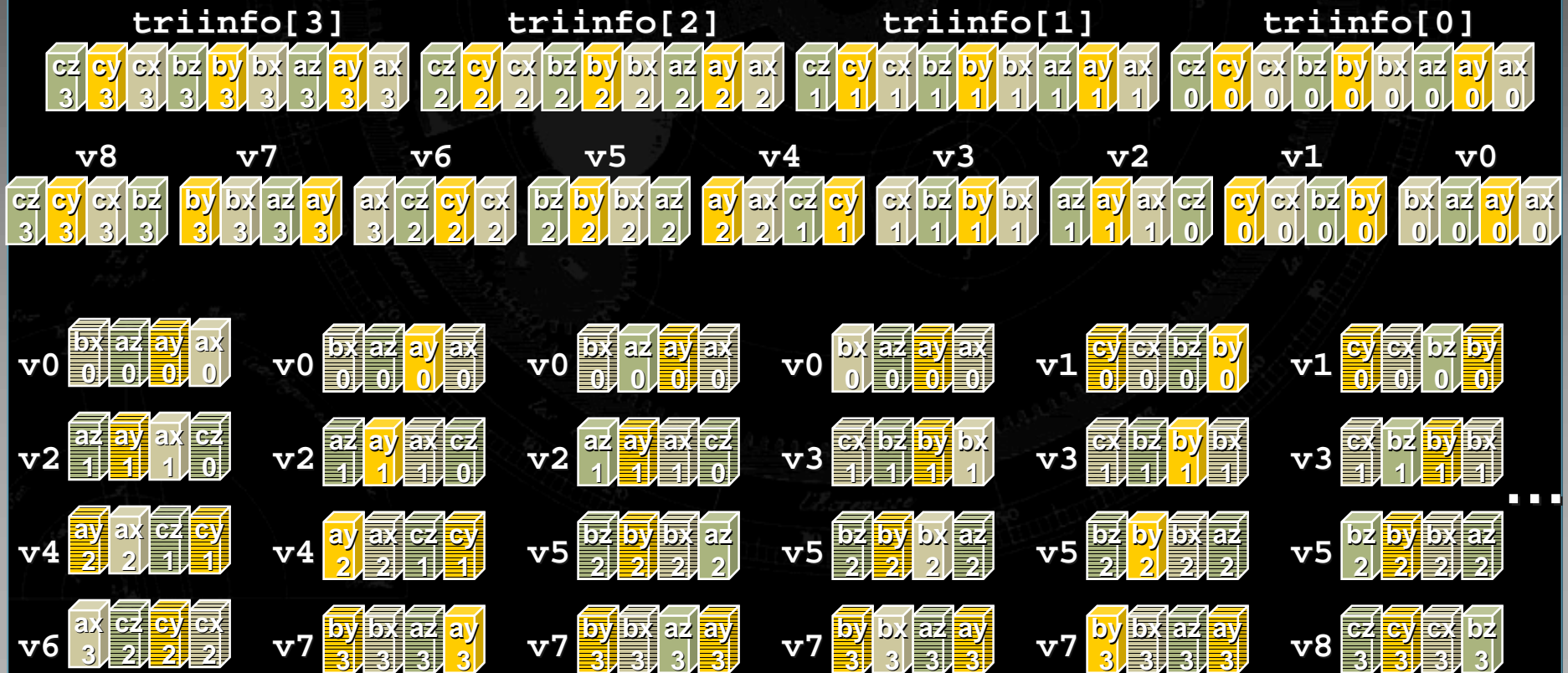


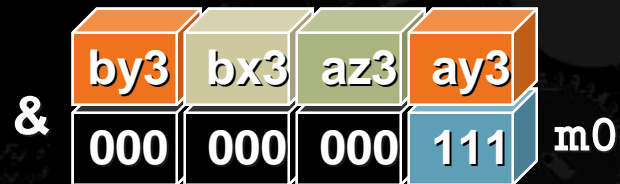
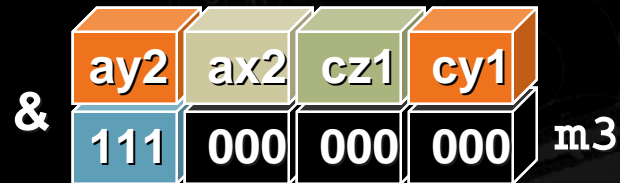
- Task for **TRIINFOtoF32vec4()**: restructure (“swizzle”) the data in minimal number of operations and preventing memory stalls

SSE helps implement optimal data swizzling!

Solving Data Swizzling Problem with SSE

Treat four `trinfos` as nine SSE operands
Combine and reshuffle matching elements





Combining and Reshuffling

AND with masks m0..m3

OR the results

Reshuffle to right order

```
ay = v[0] & m1 | v[2] & m2
      | v[4] & m3 | v[7] & m0;
RESHUFFLE(ay, 0,3,2,1);
```

Compact notation using
Vector Classes!

Step 4: Implementing Data Swizzling with SSE

- Work on function **TRIINFOtoF32vec4()**
 - a. Define masks `m0..m3` using `CONST_INT32_PS` macro
 - b. Cast array of `TRIINFOs` to array of `F32vec4s`
 - c. Mask out and combine matching elements
 - d. Reshuffle the results to the right order (where needed)
- Build, run, compare performance
- Try defining `INT32_PS` constants as static

Our Takeaway from Preparing Data with SSE

- Use SSE/SSE2-friendly data structure
- To modify data structure on-the-fly, use SSE/SSE2
- SSE/SSE2 Vector Classes are ideal to code arithmetical and logical operation

Our Agenda

Port x87-intensive code to SSE FP

Prepare data for SSE with SSE

[De]Compress data with SSE2

Summary

Solving Data Amount Problem

How to reduce memory consumed by the facet normal components?

Special property of normal components:

$$\text{abs}(nx, ny, nz) \leq 1.0$$

They can be easily mapped to **short** range!

```
● inline short float2short(float f) // scale & compact  
● {  
●     return (short)round(f * SHRT_MAX);  
● }
```

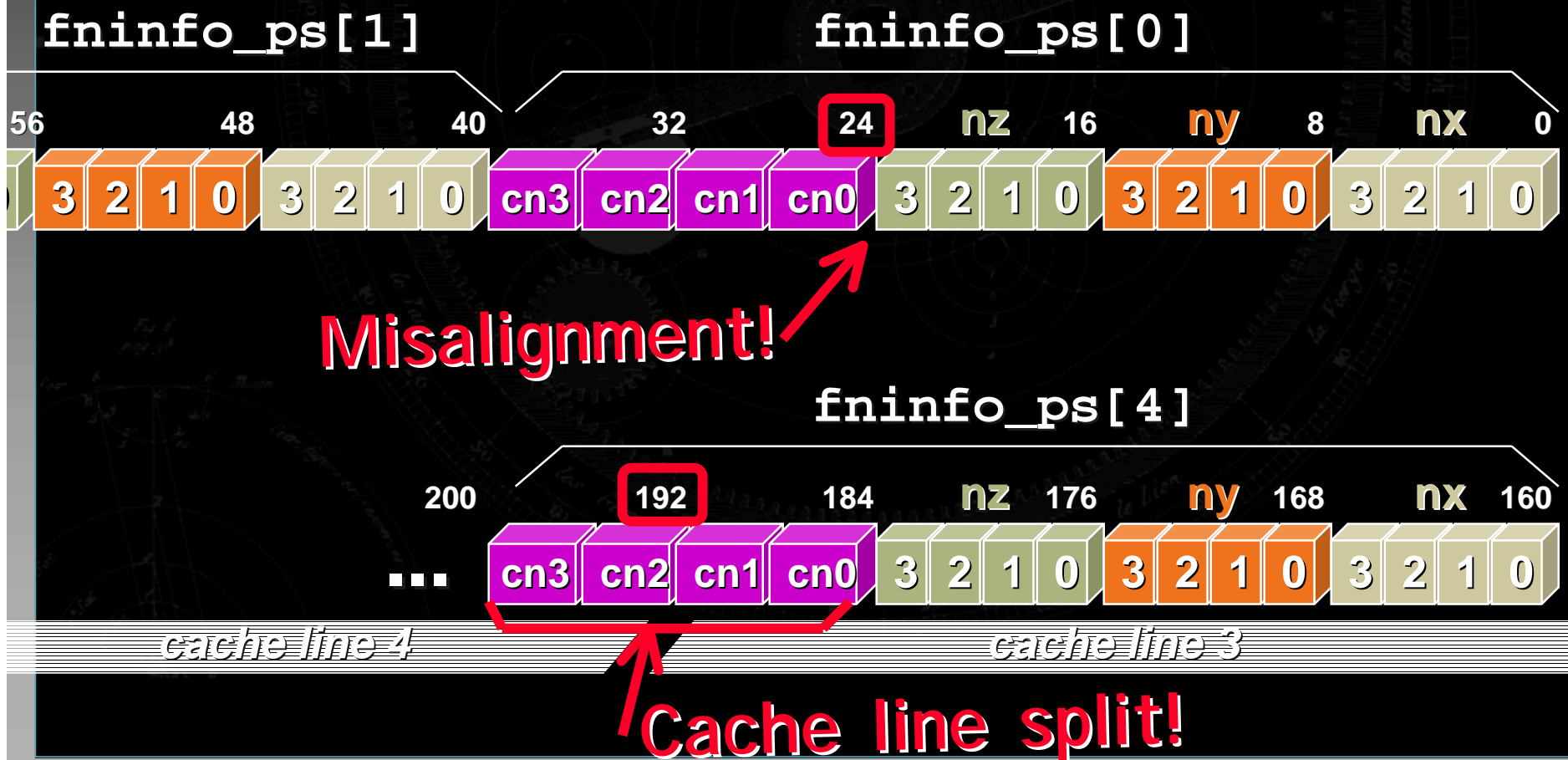
This saves 50% of storage space!

But how will we get our floats back?

With SSE2, of course!

Changing the Data Structure to Use shorts

Departure from 16n struct size can cause misalignment and cache line splits





Step 5: Data Structure for Facet Normals, Padded

Follow comments starting with `// ###`

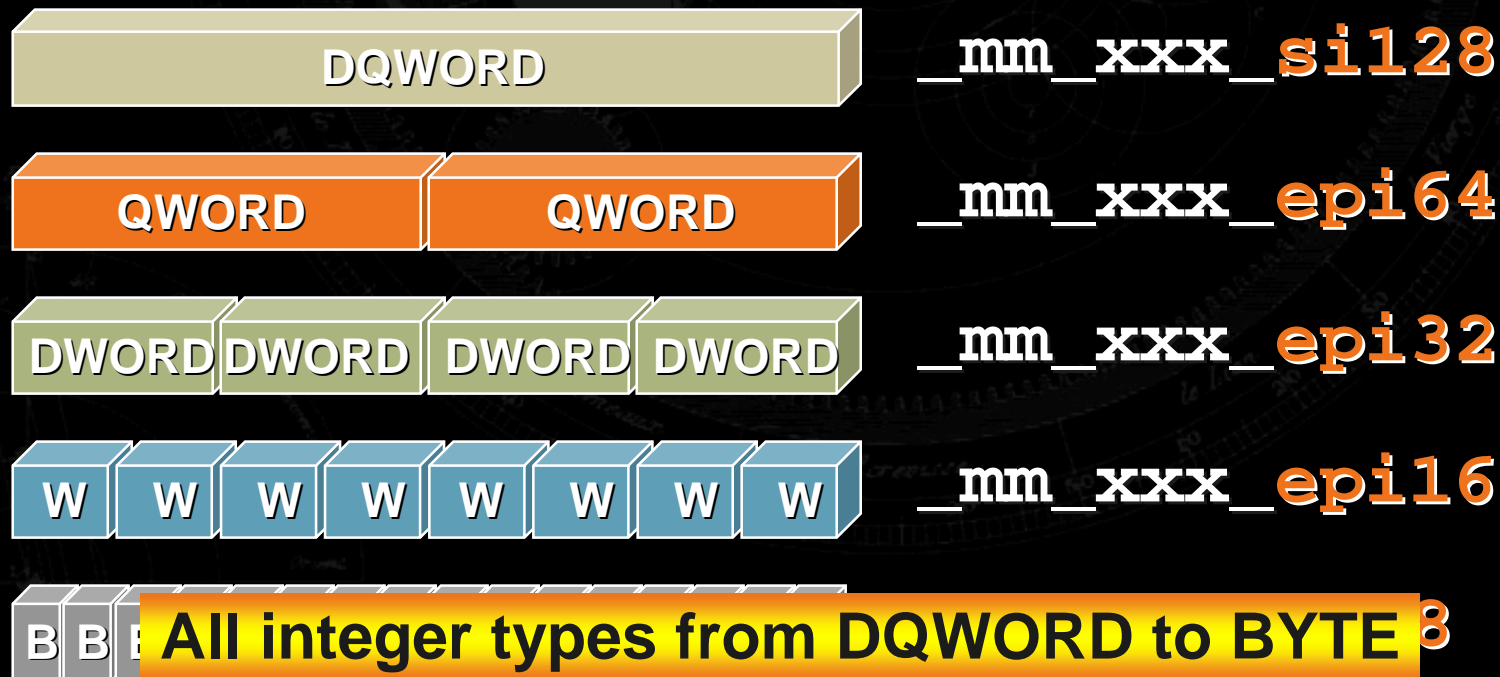
- Implement new version of `FNINFO_PS` to use packed `shorts`
- To avoid misalignments and cache line splits, add padding



Restoring structure size to 16n

SSE2 Integer 128-bit Types and Intrinsics

- All types are architecturally equivalent and freely interchangeable
- One intrinsic data type `__m128i`
- Operation type defined by intrinsic's suffix





Step 6: Converting Packed floats to Packed shorts

Inventory

- There is a conversion 4 32-bit floats \rightarrow 4 ints
- `__m128i` data type holds 8 shorts

Action Plan

1. convert packed floats (nx,ny,nz) to packed ints (inx,iny,inz)
2. pack 4+4 ints (inx,iny) into 8 shorts and store
3. pack inz with itself and store low 4 shorts

Implement in `prepfinfo_ps_B()`

Intrinsics are more flexible than Vector Classes when dealing with multiple SSE/SSE2 data types

Restoring Packed floats from Packed shorts, Plan

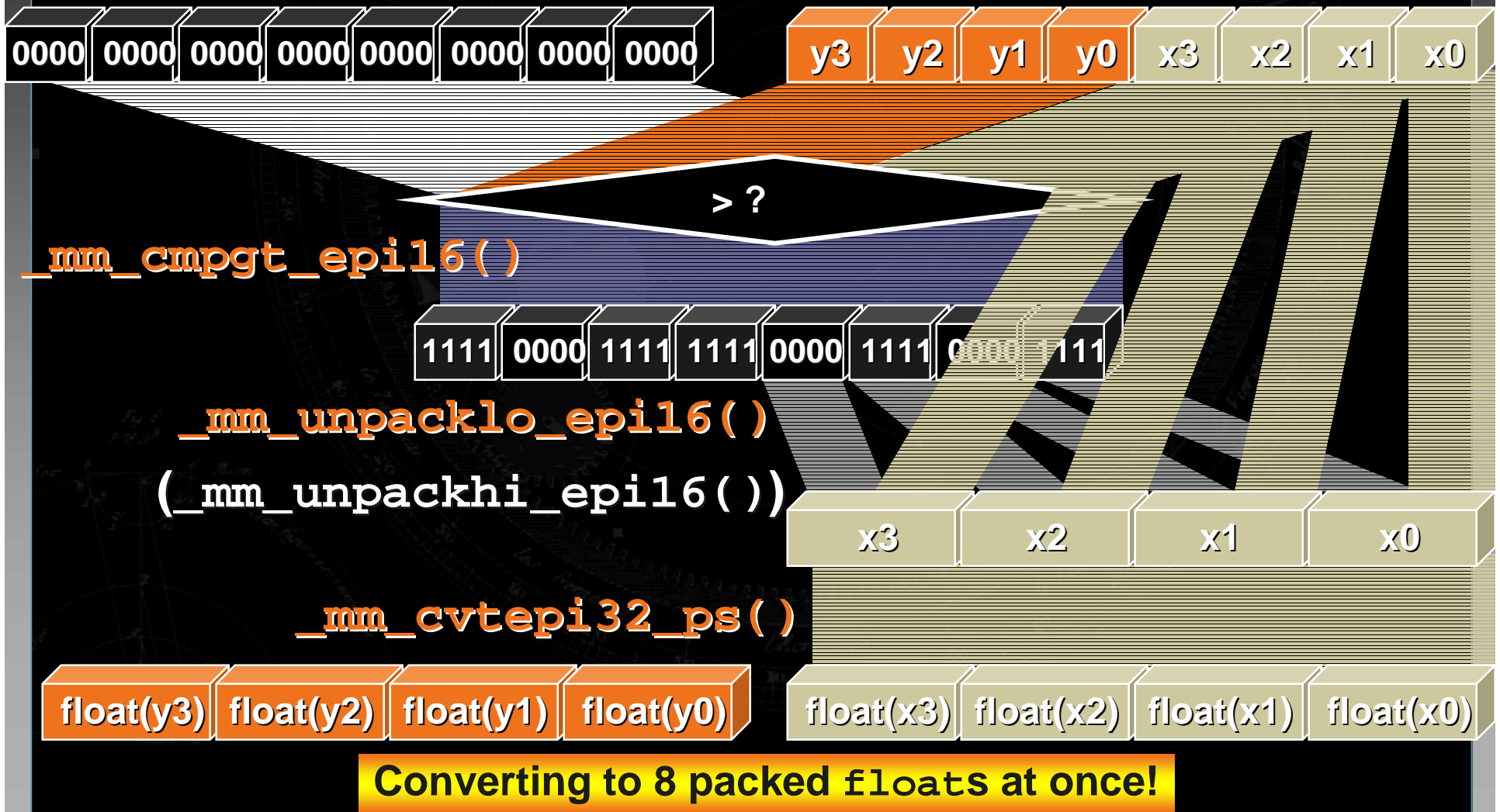
Inventory

- `__m128i` data type holds 8 shorts
- There is a conversion 4 32-bit ints \rightarrow 4 floats

Action Plan

1. load 8 shorts (nx, ny)
2. unpack (**sign-extend**) lower 4 shorts to 4 ints
3. convert to 4 floats (nx)
4. unpack higher 4 shorts to 4 ints
5. convert to 4 floats (ny)
6. load last 4 shorts (nz) -- `__mm_loadl_epi64()`
7. unpack to 4 ints, convert to 4 floats (nz)

Restoring Packed floats from Packed shorts



Rescaling

Now that we have our `floats` back...
Should we rescale every `nx`, `ny`, `nz` by
`1.0f / SHRT_MAX` ?

Suggest a better solution that

- takes care of rescaling
- doesn't require multiplication for every normal
- Hint: Look at the cull test algorithm, the dot product formula



Step 7: Restoring Packed floats

Follow comments starting with `// ###`

- a. Define a zero `__m128i` constant
- b. Define a rescaling const, `1.0f/SHRT_MAX`
- c. Rescale eye vector components
- d. Declare `__m128i` variables
- e. Load 8+4 packed short components
- f. Unpack to ints with sign-extension
- g. Convert to floats

Build (F7) and run (Ctrl+F5)

Our Takeaway from [De]Compressing Data

- For ultimate performance, look for a way to compress your data
- For on-the-fly [de]compression, use SSE2
- For rapid port and development, use Vector Classes
- For data manipulation and conversion, use SSE/SSE2 Intrinsics

Our Agenda

Port x87-intensive code to SSE FP

Prepare data for SSE with SSE

[De]Compress data with SSE2

Summary

What If...

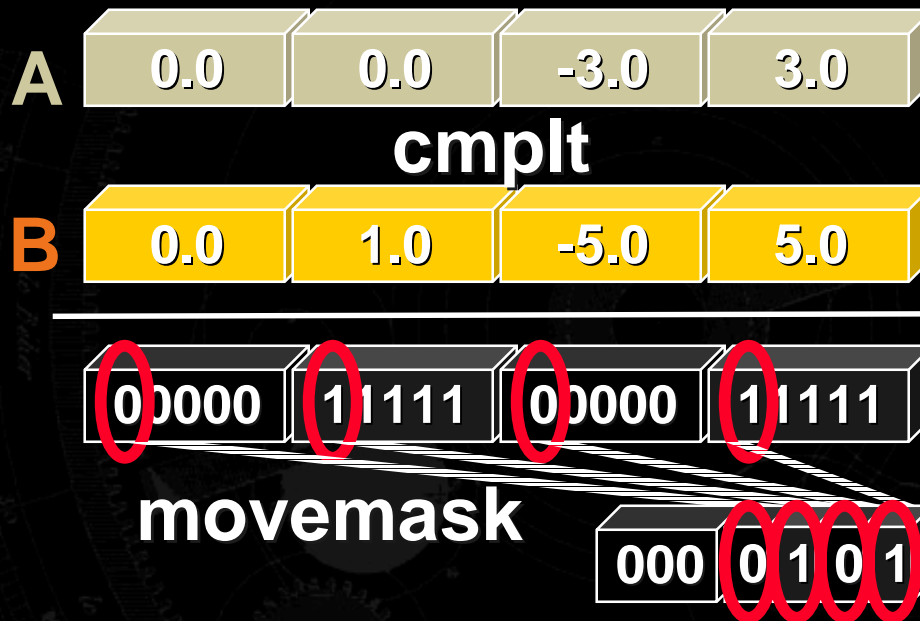
What if we were to save triangle usage flags as:

- shorts?
- BYTES?
- bits?

How could we keep count of triangles that passed the test?

Propose a data structure for compressed facet normals and $C \cdot N$ that doesn't need padding

Mask Hashing: Movemask



```
F32vec4 mask = cmplt(a, b);
int bithash = move_mask(mask);
// bithash = 0..15
switch (bithash) {
    case 0: // handle f-f-f-f
    case 1: // handle f-f-f-t
    ....   // handle other cases
    case 15: // handle t-t-t-t
}
```

Test Passed Counting



```
static const int bitcount[16] = {  
    0, // 0 == 0000  
    1, // 1 == 0001  
    1, // 2 == 0010  
    2, // 3 == 0011  
    ...  
    4 // 15 == 1111  
};  
F32vec4 mask = cmplt(a, b);  
npassed = bitcount[move_mask(mask)];
```

Our Summary

- Get performance boost with SSE / SSE2 !
- Port x87 and MMX™ code to SSE / SSE2 !
- For FP code, use SSE / SSE2 FP (both single and double precisions available!)
- For integer code, use SSE2 Integer
- For rapid port, use Vector Classes, for data manipulation and conversion, use Intrinsics
- Employ fast data [de]compression with SSE / SSE2!
- Let Intel® Compiler assist you!

SSE/SSE2 Toolbox Solutions for Real-Life SIMD Problems



Alex.Klimovitski@intel.com
Tools & Technologies Europe
Intel Corporation

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

Bonus Foils

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

Bonus Foils

Introducing SIMD: Single Instruction, Multiple Data

Scalar processing

- traditional mode
- one operation produces one result

X

+

Y

X + Y

X

Y

X + Y

SIMD processing

- with SSE / SSE2
- one operation produces multiple results

x3 x2 x1 x0

+

y3 y2 y1 y0

x3+y3 x2+y2 x1+y1 x0+y0

SSE / SSE2 SIMD Data Types

Anything that fits into 16 byte!



4x floats



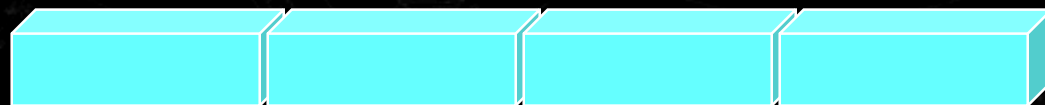
2x doubles



16x bytes



8x words



4x dwords



2x qwords



1x dqword

Matrix by Vector Example



For X, Y, Z, W:

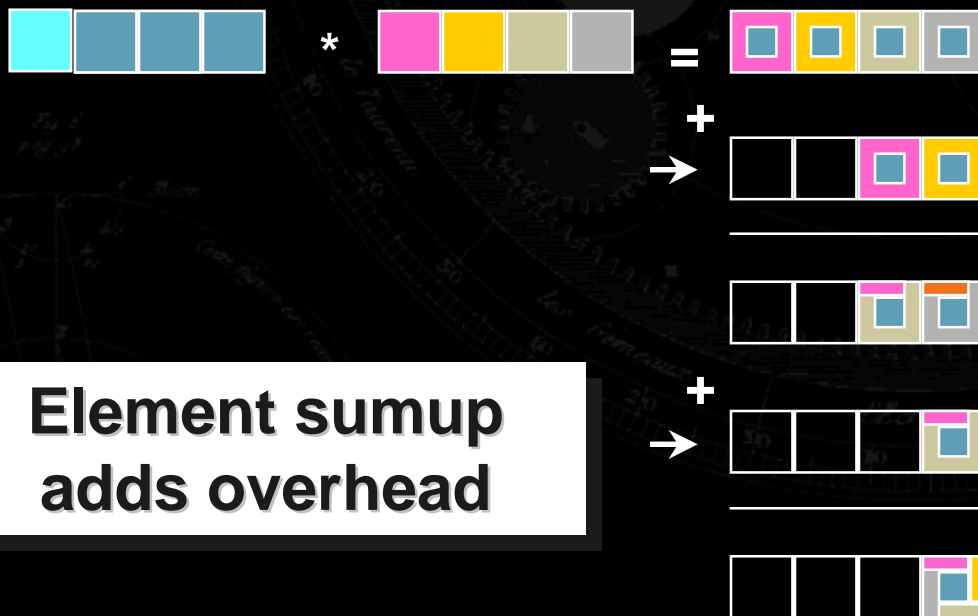
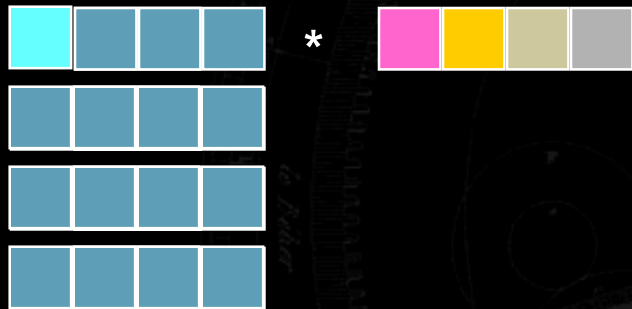


M by V Code

```
static float m[4][4];

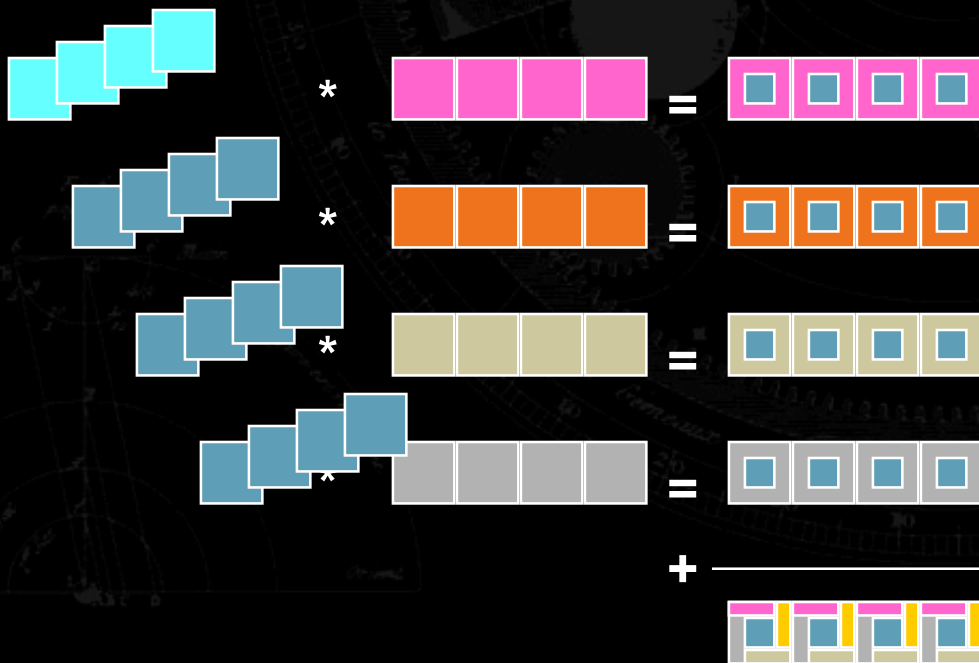
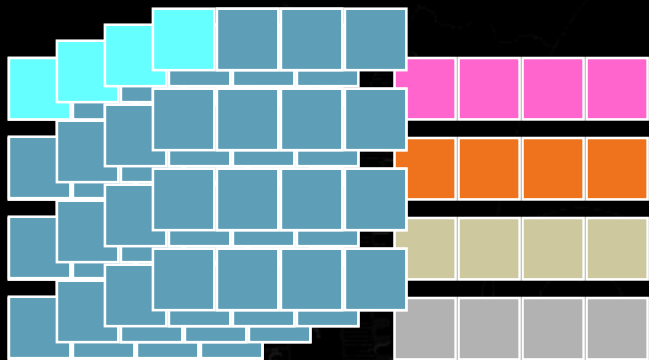
for (int i = 0; i < ARRAY_COUNT; i++) {
    float x = xi[i];
    float y = yi[i];
    float z = zi[i];
    float w = wi[i];
    xo[i] = x * m[0][0] + y * m[0][1] + z * m[0][2] +
            w * m[0][3];
    yo[i] = x * m[1][0] + y * m[1][1] + z * m[1][2] +
            w * m[1][3];
    zo[i] = x * m[2][0] + y * m[2][1] + z * m[2][2] +
            w * m[2][3];
    wo[i] = x * m[3][0] + y * m[3][1] + z * m[3][2] +
            w * m[3][3];
}
```

M by V with SSE, 1st Try



**Element sumup
adds overhead**

M by V with SSE, 2nd Try



Same Operation - Just Four at A Time!



For X, Y, Z, W:



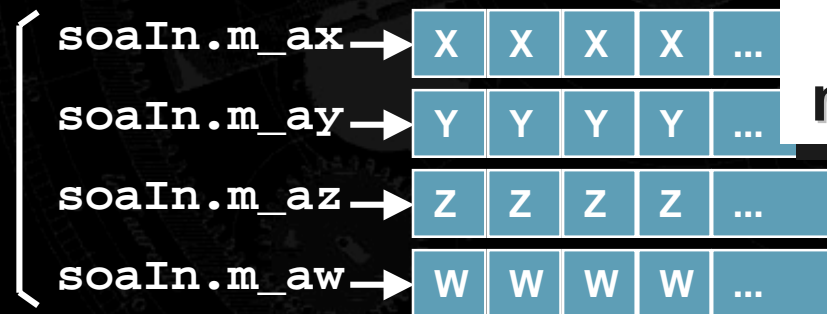
Remember Memory

AoS: Array of Structures



AoS defeats SIMD

SoA: Structure of Arrays



SoA provides for maximum parallelism!

Hybrid Structure

X4Y4Z4W4



Hybrid is also more memory-friendly!

Remember Alignment, too!

SSE/SSE2 loads/store expect data aligned on 16-byte boundary; otherwise crash!

There are unaligned load/store versions, but these are significantly slower



- `__declspec(align(16)) float a[N]; // static or auto`
- `int* b = _mm_malloc(N * sizeof(int), 16); // dynamic`
- `_mm_free(b);`
- `F32vec4 c[N / 4]; // Vec Classes are always aligned`

M by V Code with SSE

```
static F32vec4 q[4][4];

for (int i = 0; i < ARRAY_COUNT; i += 4) {
    F32vec4 x = (F32vec4&)xi[i];
    F32vec4 y = (F32vec4&)yi[i];
    F32vec4 z = (F32vec4&)zi[i];
    F32vec4 w = (F32vec4&)wi[i];
    (F32vec4&)xo[i] = x * q[0][0] + y * q[0][1] +
        z * q[0][2] + w * q[0][3];
    (F32vec4&)yo[i] = x * q[1][0] + y * q[1][1] +
        z * q[1][2] + w * q[1][3];
    (F32vec4&)zo[i] = x * q[2][0] + y * q[2][1] +
        z * q[2][2] + w * q[2][3];
    (F32vec4&)wo[i] = x * q[3][0] + y * q[3][1] +
        z * q[3][2] + w * q[3][3];
}
```

Same Code as Scalar - Just Four at A Time!

```
static float m[4][4];

for (int i = 0; i < ARRAY_COUNT; i++) {
    float x = xi[i];
    float y = yi[i];
    float z = zi[i];
    float w = wi[i];
    xo[i] = x * m[0][0] + y * m[0][1] + z * m[0][2] +
           w * m[0][3];
    yo[i] = x * m[1][0] + y * m[1][1] + z * m[1][2] +
           w * m[1][3];
    zo[i] = x * m[2][0] + y * m[2][1] + z * m[2][2] +
           w * m[2][3];
    wo[i] = x * m[3][0] + y * m[3][1] + z * m[3][2] +
           w * m[3][3];
}
```

M by V with Perspective Correction Code

```
for (int i = 0; i < ARRAY_COUNT; i++) {  
    float x = xi[i];  
    float y = yi[i];  
    float z = zi[i];  
    float w = wi[i];
```

```
    float wr = 1.0 / (x * m[3][0] + y * m[3][1] +  
        z * m[3][2] + w * m[3][3]);
```

```
    xo[i] = wr * (x * m[0][0] + y * m[0][1] +  
        z * m[0][2] + w * m[0][3]);  
    yo[i] = wr * (x * m[1][0] + y * m[1][1] +  
        z * m[1][2] + w * m[1][3]);  
    zo[i] = wr * (x * m[2][0] + y * m[2][1] +  
        z * m[2][2] + w * m[2][3]);  
    wo[i] = wr;
```

```
}
```

M by V with Perspective Correction SSE Code

```
for (int i = 0; i < ARRAY_COUNT; i += 4) {
```

```
    F32vec4 x = (F32vec4&)xi[i];
```

```
    F32vec4 y = (F32vec4&)yi[i];
```

```
    F32vec4 z = (F32vec4&)zi[i];
```

```
    F32vec4 w = (F32vec4&)wi[i];
```

```
    F32vec4 wr = rcp_nr(x * q[3][0] + y * q[3][1] +  
        z * q[3][2] + w * q[3][3]);
```

```
    (F32vec4&)xo[i] = wr * (x * q[0][0] + y * q[0][1]  
        + z * q[0][2] + w * q[0][3]);
```

```
    (F32vec4&)yo[i] = wr * (x * q[1][0] + y * q[1][1]  
        + z * q[1][2] + w * q[1][3]);
```

```
    (F32vec4&)zo[i] = wr * (x * q[2][0] + y * q[2][1]  
        + z * q[2][2] + w * q[2][3]);
```

```
    (F32vec4&)wo[i]
```

```
}
```

Easy per-component processing!

“SIMDizing” The Matrix

```
● void FourFloats2F32vec4(F32vec4* v, const float* f) ●
● { ●
●     v[0]=_mm_load_ps(f); ●
●     v[1]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(1,1,1,1)); ●
●     v[2]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(2,2,2,2)); ●
●     v[3]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(3,3,3,3)); ●
●     v[0]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(0,0,0,0)); ●
● } ●
●
● static _MM_ALIGN16 float m[4][4]; ●
● static F32vec4 q[4][4]; ●
●
● for (int i = 0; i < 4; i++) ●
●     FourFloats2F32vec4(q[i], m[i]); ●
```

Align scalar data, too!

Rules of Good Parallelism

- Maintain the original algorithm
- Process {four} data portions in parallel
- Keep only homogeneous components in one SIMD operand
- {Quadruple} loop-invariants outside the loop to create SIMD invariants
- Use SIMD-friendly structure, SoA or Hybrid

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

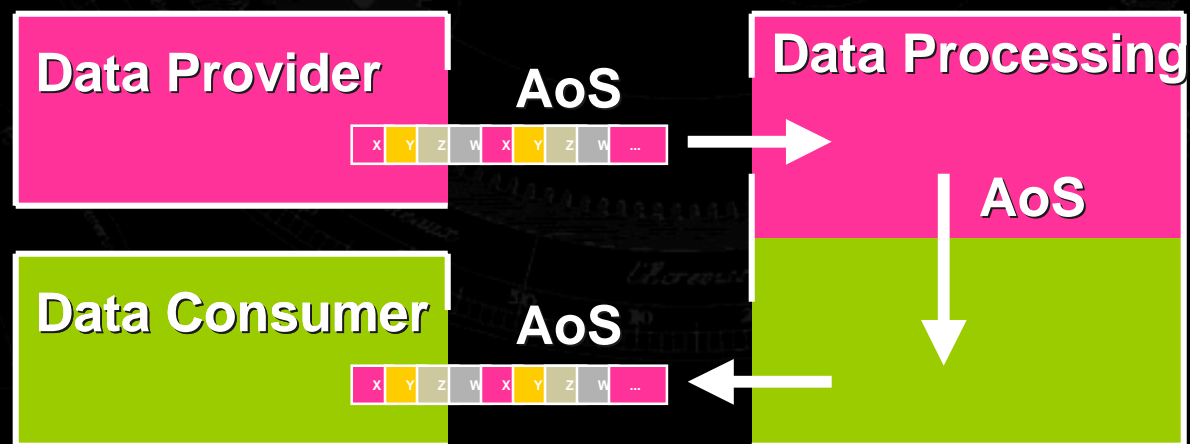
Bonus Foils

SIMD-Unfriendly Data Structures

The primary SIMD problem

Results from:

- Interface / API Constraints
- Algorithm Logic
- Legacy Code

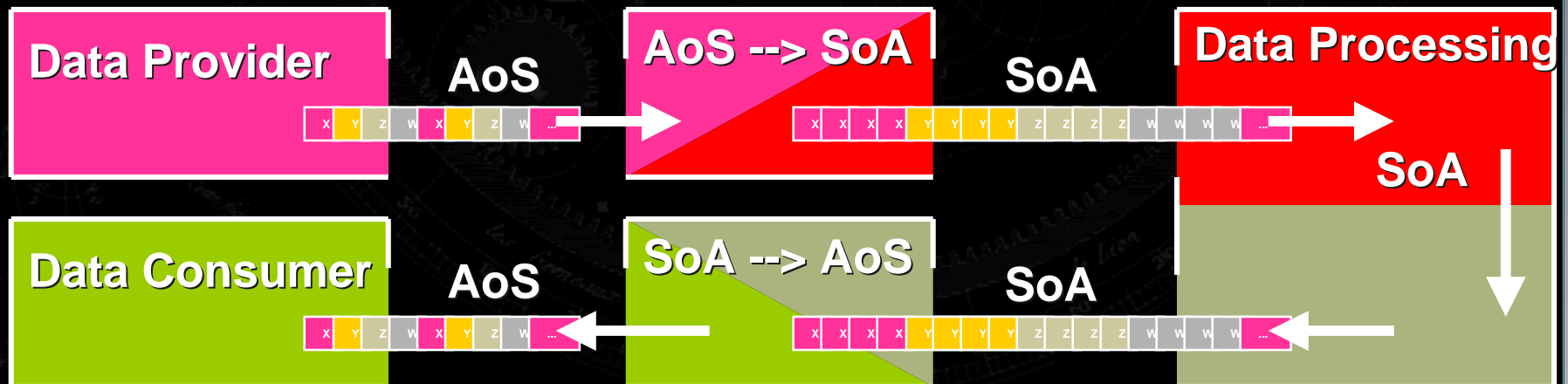


Taming SIMD-Unfriendly Data

“Swizzle” (transform) data at run-time

Pre-swizzle at design/load-time as much as possible

Implement swizzle with SSE / SSE2



Data Swizzling Wrong Way

- F32 vec4 x, y, z, w;
- float temp[4];
- temp[0] = aosIn[0].x;
- temp[1] = aosIn[1].x;
- temp[2] = aosIn[2].x;
- temp[3] = aosIn[3].x;
- x = _mm_load_ps(temp);
- // same for y, z, w

local var-s
(Registers)

memory stall

load_ps

temp

w3 z3 y3 x3 w2 z2 y2 x2 w1 z1 y1 x1 w0 z0 y0 x0 ← aosIn

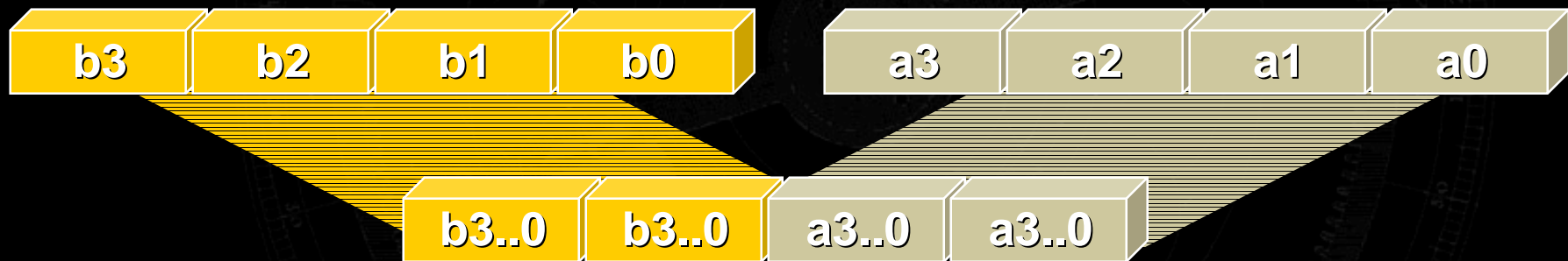
memory

Do NOT use scalar ops to prepare SIMD-friendly data!

Chief SIMD Swizzler: Shuffle

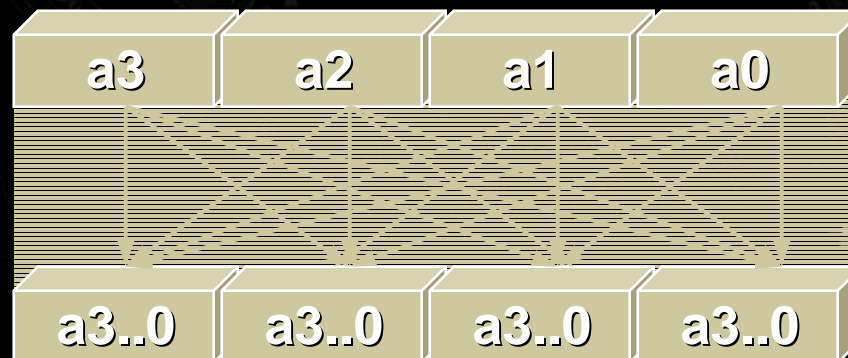
First operand contributes two lower elements,
second operand contributes two higher ones

```
_mm_shuffle_ps(a, b, _MM_SHUFFLE(3,1,2,0))
```

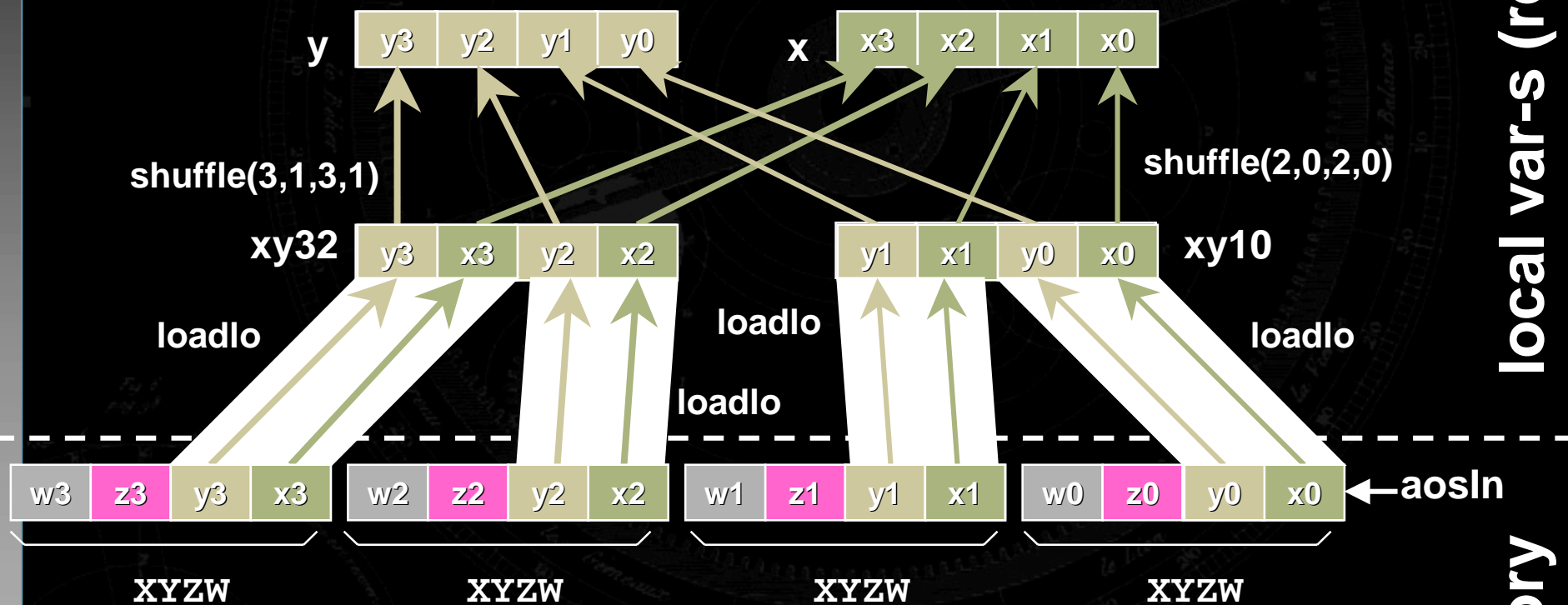


- Shuffle with itself: total swizzle

```
_mm_shuffle_ps(a, a, _MM_SHUFFLE(3,1,2,0))
```

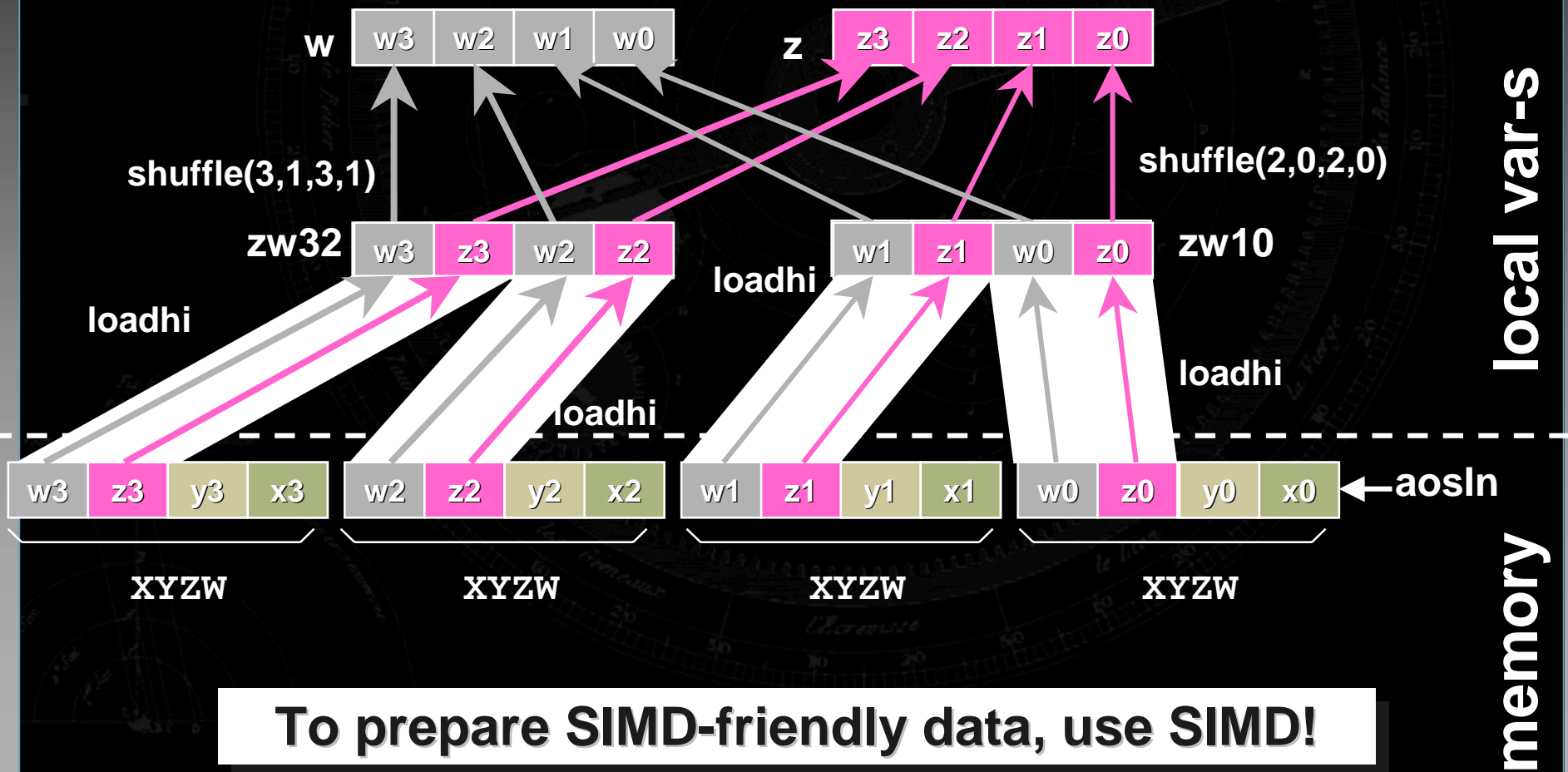


Data Swizzling with SIMD: AoS to SoA



Similar steps for z, w!

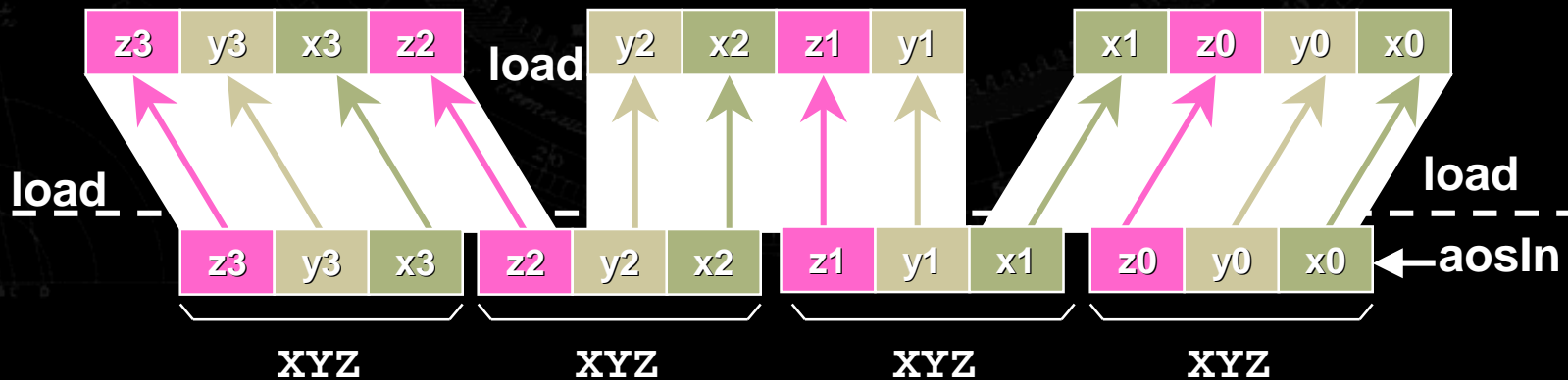
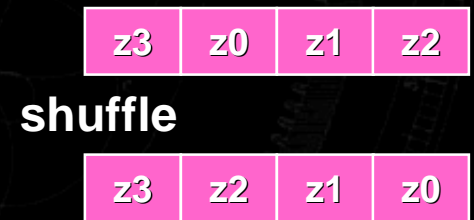
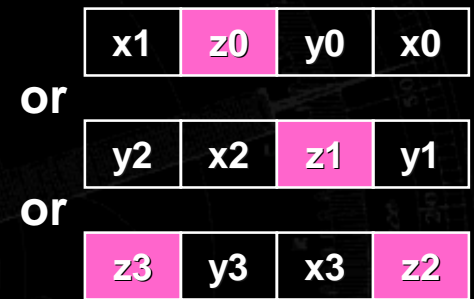
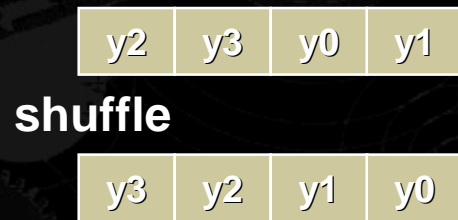
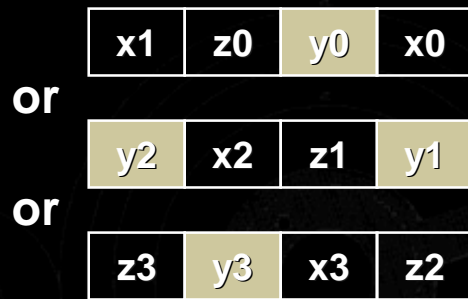
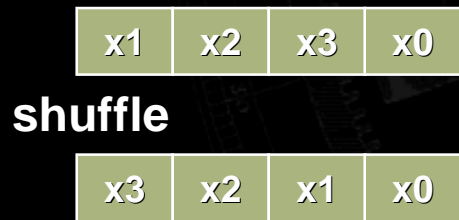
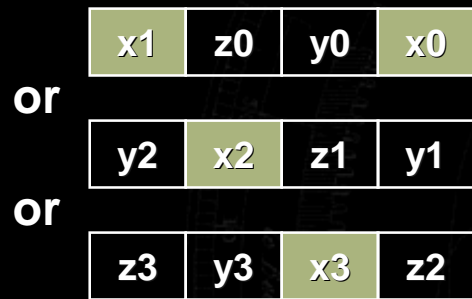
Data Swizzling with SIMD: AoS to SoA (continued)



AoS to SoA Code

```
void XYZWtoF32vec4(F32vec4& x, y, z, w, XYZW* aosIn)
{
    F32vec4 xy10, xy32, zw10, zw32;
    xy10 = zw10 = _mm_setzero_ps();
    xy10 = _mm_loadl_pi(xy10, (__m64*)&(aosIn[0]).x);
    zw10 = _mm_loadl_pi(zw10, (__m64*)&(aosIn[0]).z);
    xy10 = _mm_loadh_pi(xy10, (__m64*)&(aosIn[1]).x);
    zw10 = _mm_loadh_pi(zw10, (__m64*)&(aosIn[1]).z);
    xy32 = zw32 = _mm_setzero_ps();
    xy32 = _mm_loadl_pi(xy32, (__m64*)&(aosIn[2]).x);
    zw32 = _mm_loadl_pi(zw32, (__m64*)&(aosIn[2]).z);
    xy32 = _mm_loadh_pi(xy32, (__m64*)&(aosIn[3]).x);
    zw32 = _mm_loadh_pi(zw32, (__m64*)&(aosIn[3]).z);
    x = _mm_shuffle_ps(xy10, xy32, SHUFFLE(2,0,2,0));
    y = _mm_shuffle_ps(xy10, xy32, SHUFFLE(3,1,3,1));
    z = _mm_shuffle_ps(zw10, zw32, SHUFFLE(2,0,2,0));
    w = _mm_shuffle_ps(zw10, zw32, SHUFFLE(3,1,3,1));
}
```

3-Component AoS to SoA



Defining SSE Bit Masks

- No valid `float` with bit pattern needed?
- Define aligned static array of four integers
- Load it at runtime as packed `floats`
- Implemented as a macro `CONST_INT32_PS`

```
#define CONST_INT32_PS(N, V3,V2,V1,V0) \  
static const _MM_ALIGN16 int _##N[] = \  
    {V0, V1, V2, V3}; /*little endian!*/ \  
const F32vec4 N = _mm_load_ps((float*)_##N);  
  
// usage example, mask for elements 3 and 1:  
CONST_INT32_PS(mask31, ~0, 0, ~0, 0);
```

Swizzling 3-Component AoS to SoA Code

```
void XYZtoF32vec4(F32vec4& x, y, z, XYZ* aosIn)
{
    F32vec4 a, b, c;
    CONST_INT32_PS(mask30, ~0, 0, 0, ~0); // etc.

    a = _mm_load_ps((float*)aosIn);
    b = _mm_load_ps(((float*)aosIn) + 4);
    c = _mm_load_ps(((float*)aosIn) + 8);

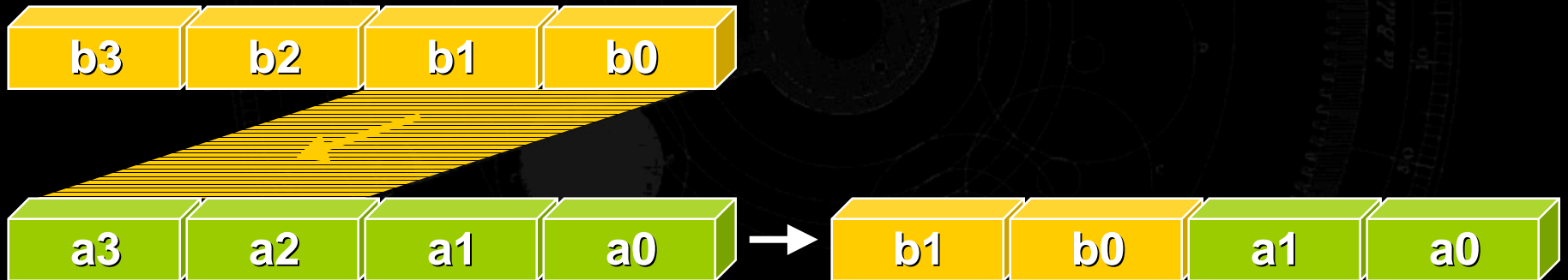
    x = (a & mask30) | (b & mask2) | (c & mask1);
    y = (a & mask1) | (b & mask30) | (c & mask2);
    z = (a & mask2) | (b & mask1) | (c & mask30);

    x = _mm_shuffle_ps(x, x, _MM_SHUFFLE(1,2,3,0));
    y = _mm_shuffle_ps(y, y, _MM_SHUFFLE(2,3,0,1));
    z = _mm_shuffle_ps(z, z, _MM_SHUFFLE(3,0,1,2));
}
```

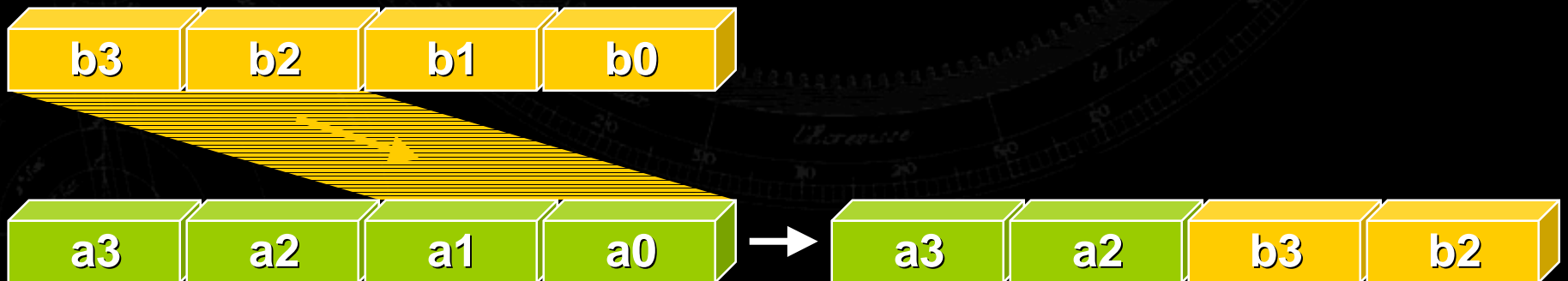
Gatherers: Cross-Half-Moves

Move lower (higher) half of the second operand to higher (lower) half of the first operand

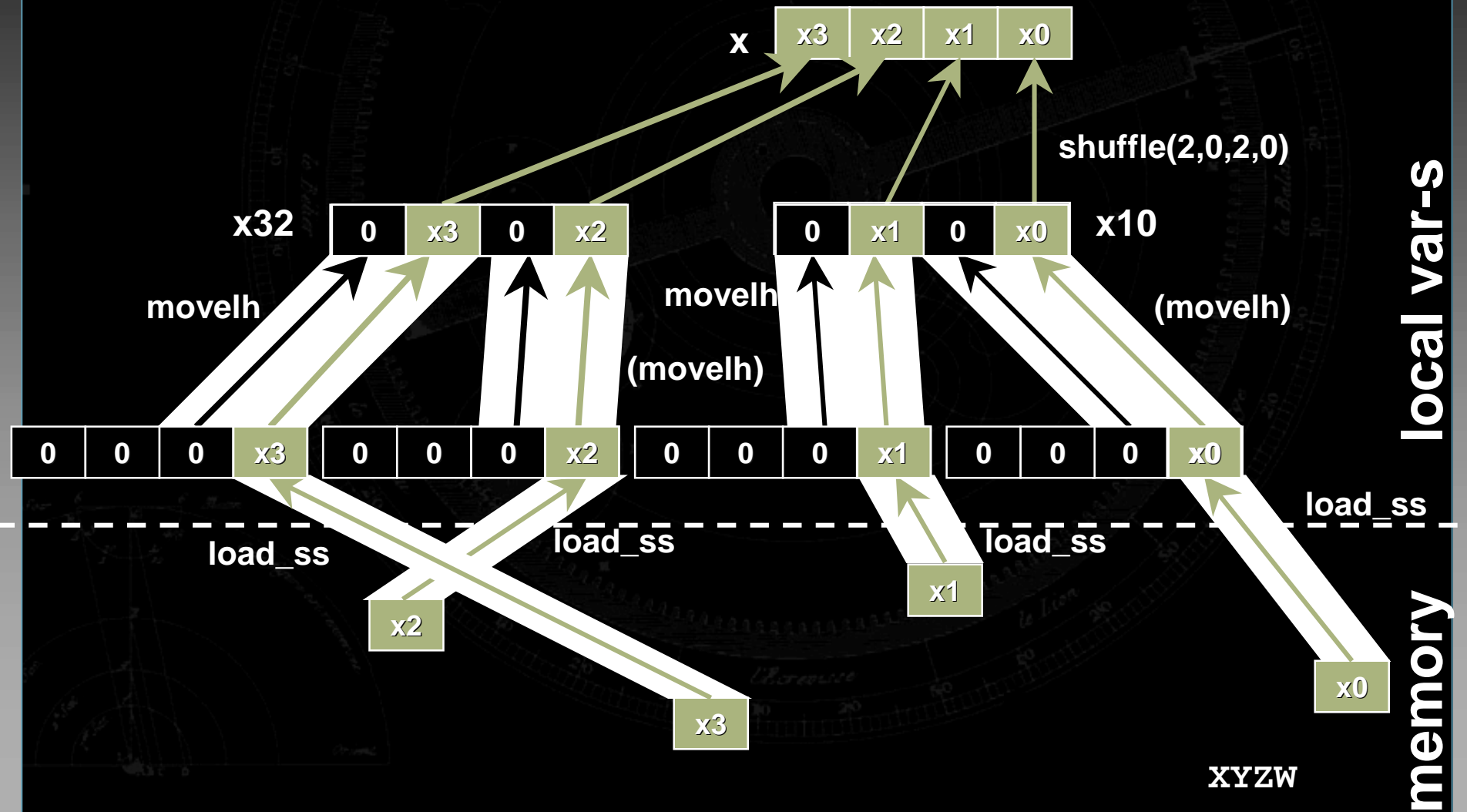
`_mm_move`**lh**_ps(*a*, *b*)



`_mm_move`**hl**_ps(*a*, *b*)



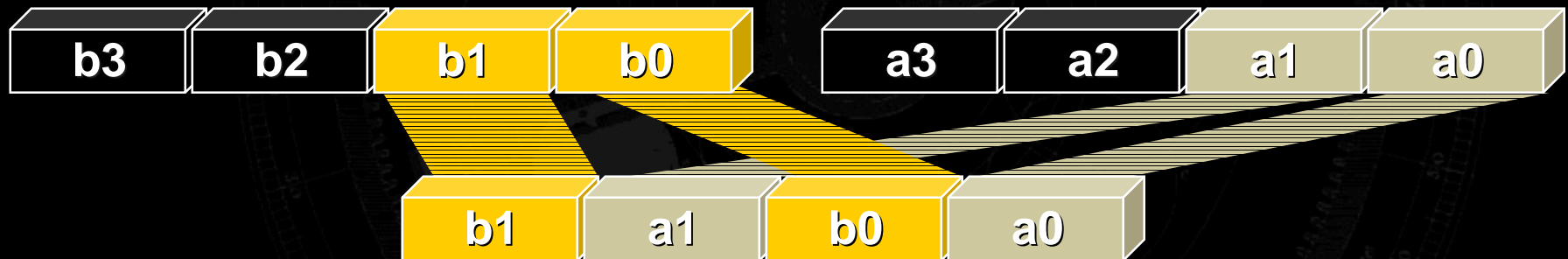
Scatter-Gathering + Swizzling



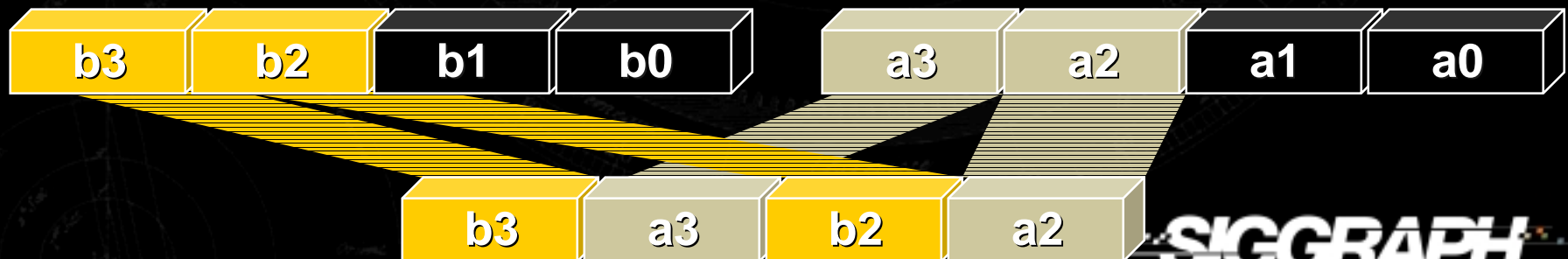
Chief Unswizzler: Unpack

Two lower(higher) elements from the first operand and two lo(hi) ones from the second are **interleaved**

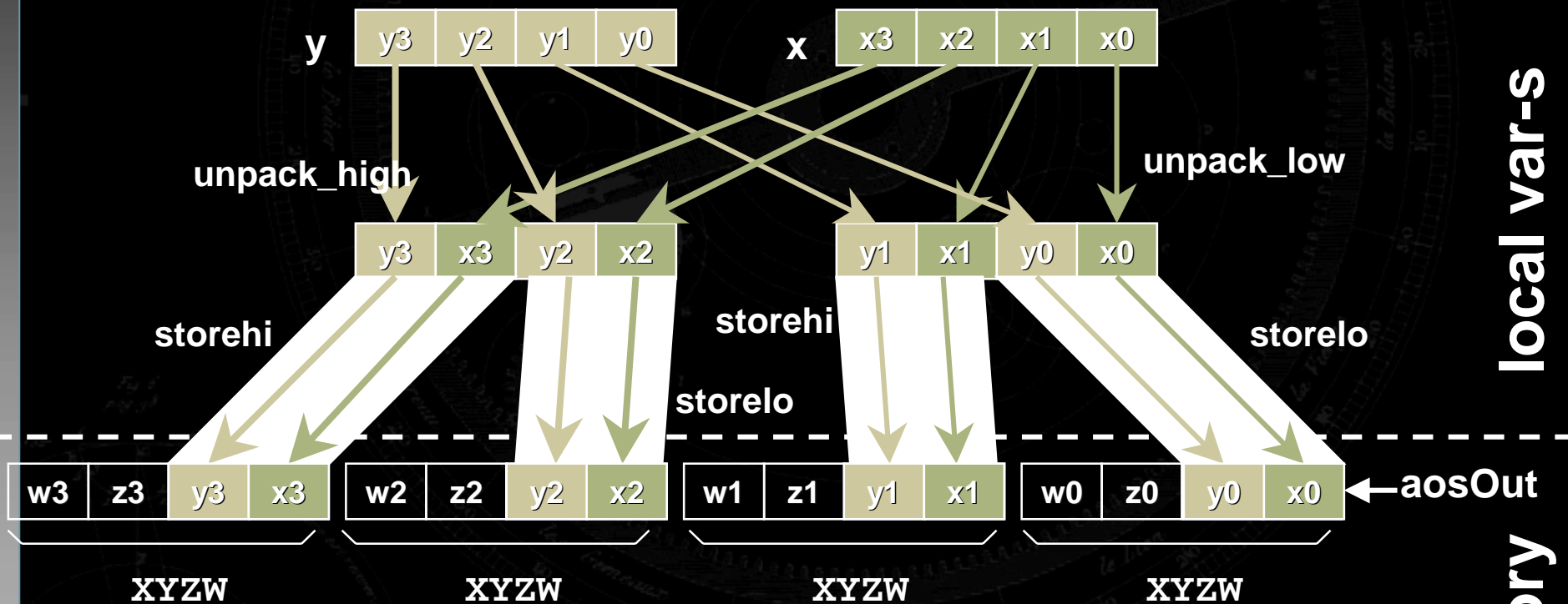
`unpack_low(a, b)`



`unpack_high(a, b)`

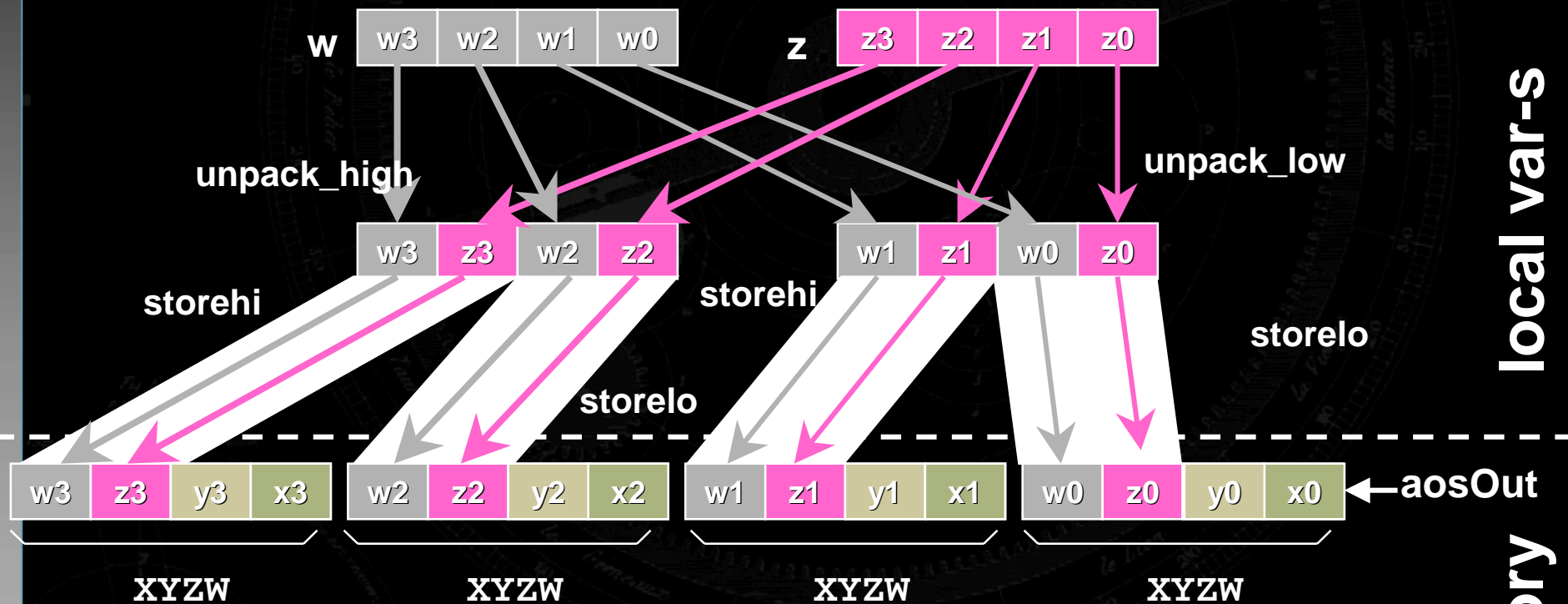


Data Unswizzling with SIMD SoA to AoS

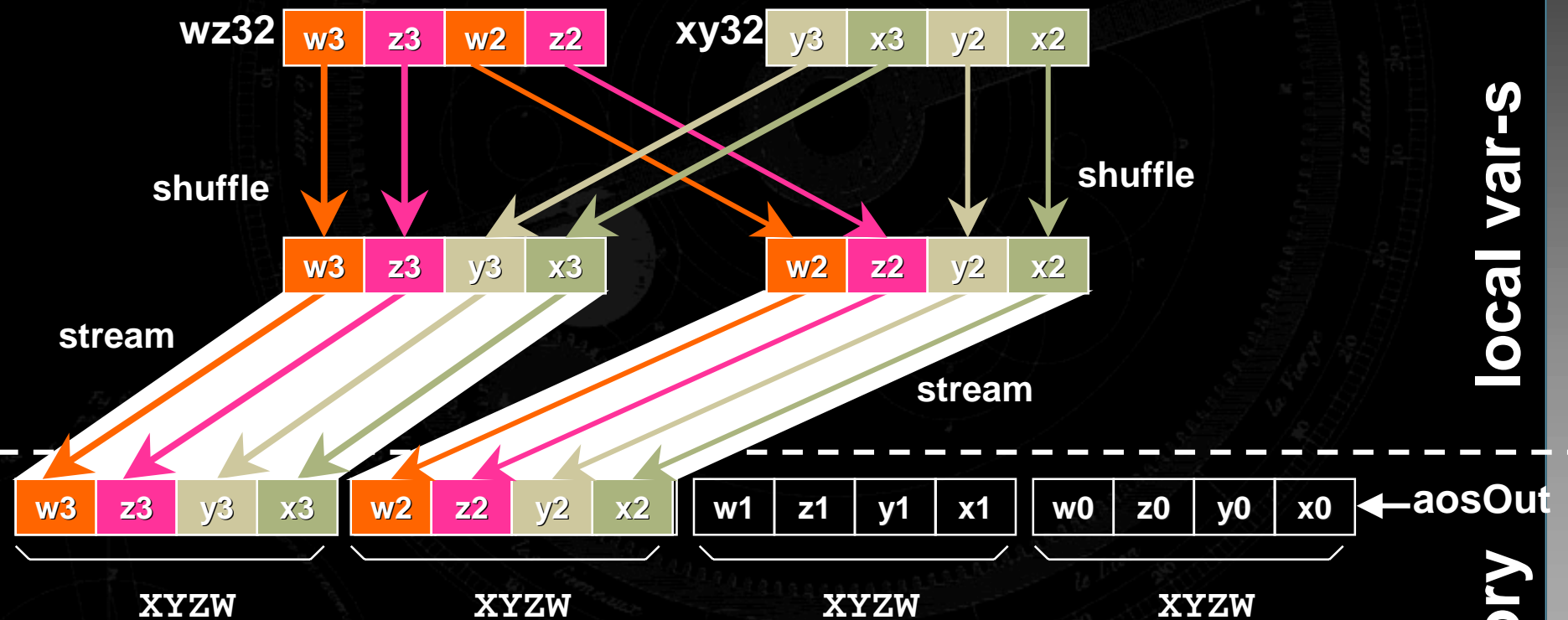


Similar steps for z, w!

Data Unswizzling with SIMD SoA to AoS (continued)



SoA to AoS with Streaming Store



Data Restructuring Summary

- Use SIMD-friendly SoA or Hybrid structures whenever possible
- Use SSE/2 to swizzle SIMD-unfriendly structures before processing
- Use SSE/2 to store results of SIMD processing into SIMD-unfriendly structures (unswizzling)
- Look for more restructuring solutions in Bonus Foils!

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

Bonus Foils

Data Compression with Integers

FP value inside a known range can be mapped into a compacter int value

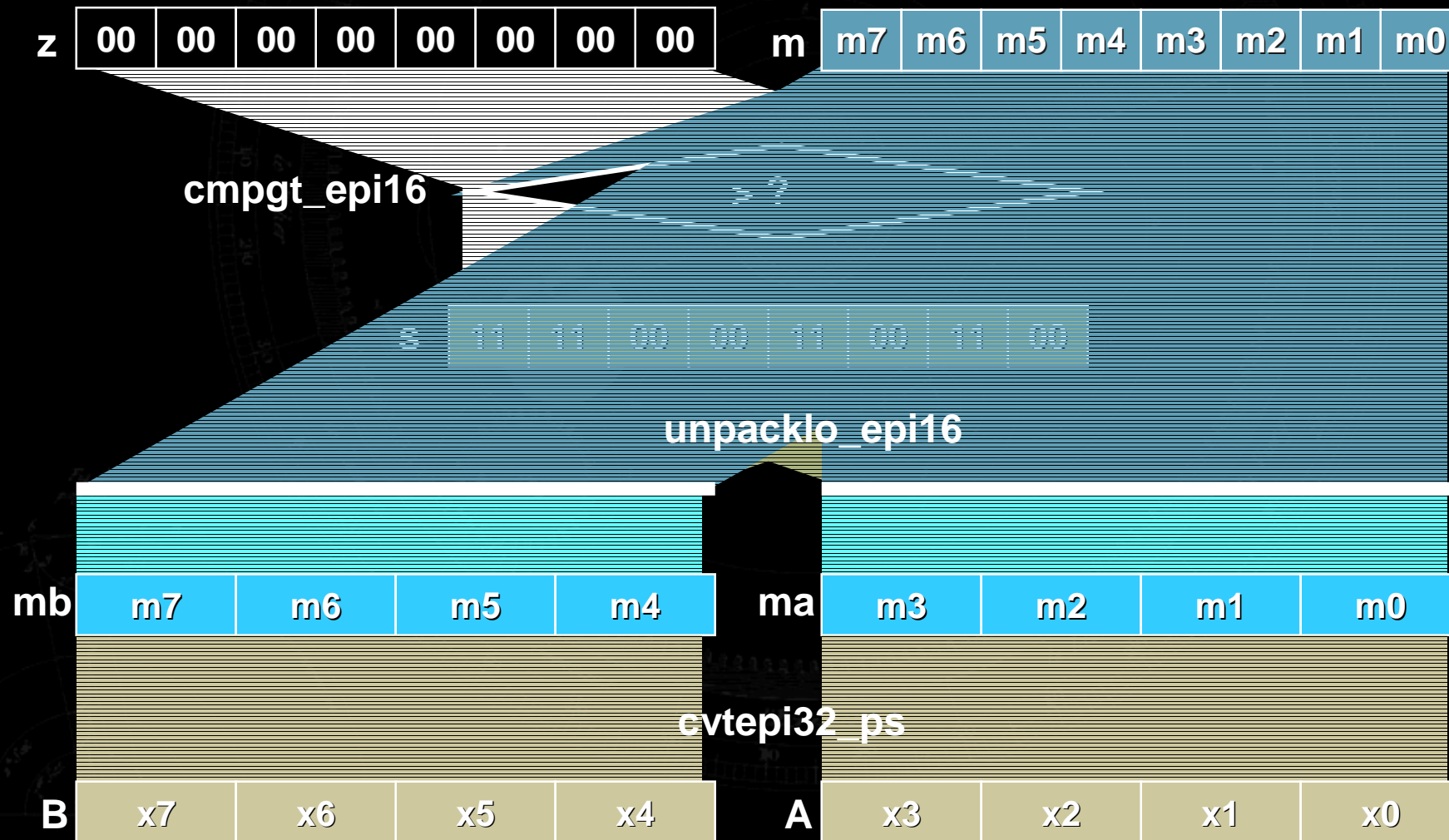
$$i = i_{\min} + \left[(f - f_{\min}) \cdot \frac{i_{\max} - i_{\min}}{f_{\max} - f_{\min}} \right]; \text{ for symmetric range: } i = \left[f \cdot \frac{i_{\text{range}}}{f_{\text{range}}} \right]$$
$$f = f_{\min} + (i - i_{\min}) \cdot \frac{f_{\max} - f_{\min}}{i_{\max} - i_{\min}}; \text{ for symmetric range: } f = i \cdot \frac{f_{\text{range}}}{i_{\text{range}}}$$

Example: -1.0..+1.0 \rightarrow -/+SHRT_MAX

```
●    short s; float f; ●  
●    s = (short)round(f * SHRT_MAX); ●  
●    f = float(s) * (1.0f / SHRT_MAX); ●
```

Application examples: facet normals, lighting normals, landscape heights...

SIMD Short → SIMD Float Conversion with SSE2



Data Compression Summary

- Save memory traffic and cache space by [de]compressing data on-the-fly
- Use SSE / SSE2 for type conversions
- Swizzle short integer data before conversion - achieve wider parallelism

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

Bonus Foils

Conditions without Branches

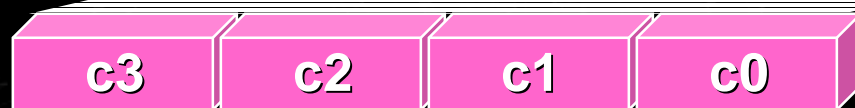
$R = (A < B) ? C : D$ //remember: everything packed



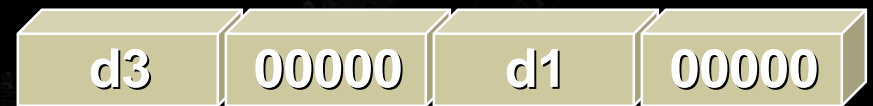
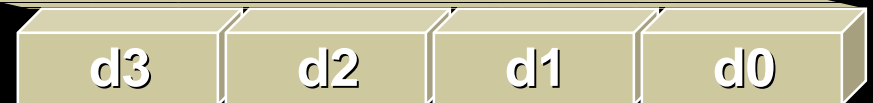
cmplt



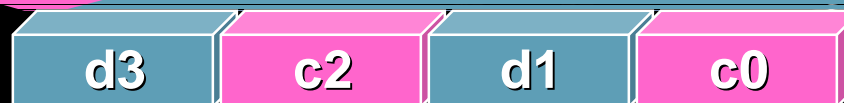
and



nand



or



Conditions without Branches Code

```
// R = (A < B)? C : D
```

```
F32vec4 mask = cmplt(a, b);
```

```
r = (mask & c) | _mm_nand_ps(mask, d);
```

```
// OR, using F32vec4 friend function:
```

```
r = select_lt(a, b, c, d);
```

Conditional Processing with SSE / SSE2

- Scalar

if (a < b)

calculate c

r = c

else

calculate d

r = d

- SSE / SSE2

mask = cmplt(a, b)

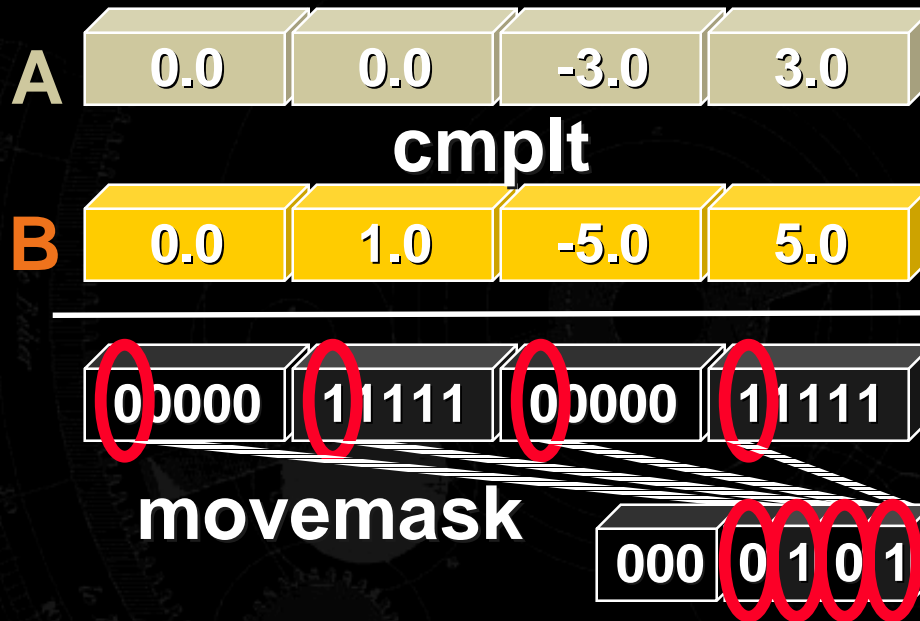
calculate c

calculate d

r = (mask & c) |
nand(mask, d)

Utilize data-level and instruction-level parallelism!

Branch Hub: Movemask



```
F32vec4 mask = cmplt(a, b);
```

```
if (move_mask(mask)) {  
    // do only if at least one is true  
    // can be logic-conditional here  
}
```

One jump is better than many!

Conditional Processing with SSE / SSE2, Movemask

- Scalar

if (a < b)

calculate c

r = c

else

calculate d

r = d

- SSE/SSE2, Movemask

mask = cmplt(a, b)

switch (move_mask(mask))

case 0xf:

calculate c

r = c

case 0x0:

calculate d

r = d

default:

calculate c
calculate d

r = (mask&c)|nand(mask,d)

SIMD for Conditional Code Summary

- You can successfully use SSE/SSE2 even with conditional, branchy code
- Replace branches with logic or computation
- Reduce total number of branches with movemask
- Look for more examples in Bonus Foils

Agenda

Exploiting Parallelism

Data Restructuring

Data Compression

Conditional Code with SIMD

Summary

Bonus Foils

What Is in Bonus Foils?

Using Automatic Vectorizer

- Compiler can do SSE/SSE2 for you!

More Conditional Code with SIMD

- Abs function, flag accumulation ("clipping"), test passed counting...

Applying SSE/SSE2 to Scalar Code

- What if algorithm is inherently scalar?
or there are no long data arrays?
- Still get performance with SSE/SSE2!

Summary: Call to Action

- Accelerate all your critical code with SSE / SSE2 processing
- Make your data SIMD-friendly
- Use SSE / SSE2 for on-the-fly data swizzling and [de]compression
- Use SSE / SSE2 comparisons & logic to replace conditional code
- Extend your own SSE/SSE2 Toolbox!

<http://developer.intel.com/design/pentium4/>
<http://developer.intel.com/IDS>

Bonus Foils

Bonus Foils

Using Automatic Vectorizer

More Conditional Code with SIMD

Applying SIMD to Scalar Code

Using Intel Compiler's Automatic Vectorizer

- Now that SSE/SSE2 is so easy, the compiler can do it for you!
- Steps to using Automatic Vectorizer:
 1. Understand for yourself how to SIMDize
 2. Prepare and align the data structures
 3. Provide hints such as unaliased pointers
 4. Invoke Automatic Vectorizer
 5. SIMDize remaining critical code with Vector Classes and Intrinsics

Invoking Automatic Vectorizer

-O2 -QaxW -Qvec_report3

-O2 “optimize for speed”

- standard Visual C++* Release build setting

-QaxW “optimize using SSE and SSE2”

- also invokes Automatic Vectorizer
- auto-versions optimized code for compatibility
- ignored by Microsoft* C++ compiler

-Qvec_report3 “report on vectorization”

See Intel Compiler documentation for more power options!

Automatic Vectorizer in Action

```
● void MbyV(float* xi, float* yi, float* zi, float* wi, ●  
● float* restrict xo, float* restrict yo, ●  
● float* restrict zo, float* restrict wo) ●  
● { ●  
●   __assume_aligned(xi, 16); ... // same for yi,zi,wi ●  
●   for (int i = 0; i < ARRAY_COUNT; i++) { ●  
●     float x = xi[i]; float y = yi[i]; ●  
●     float z = zi[i]; float w = wi[i]; ●  
●     wr = 1.0 / (x * matrix[3][0] + y * matrix[3][1] + ●
```

Classview | Fileview

```
-----Configuration: AoSSoA - Win32 Release-----  
x  
Compiling...  
icl AoSSoA.cpp  
C:\users\IdfSpring99\Lab1\AoSSoA.cpp(68) : (col. 2) remark: LOOP WAS VECTORIZED.  
AoSSoA.obj - 0 error(s), 0 warning(s)
```

Build

Debug

Find in Files 1

Fin

Bonus Foils

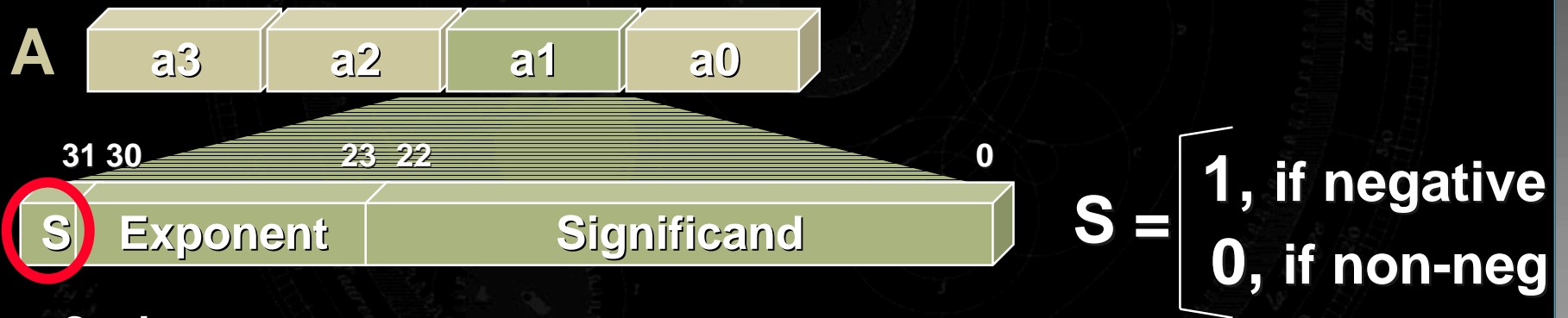
Using Automatic Vectorizer

More Conditional Code with SIMD

Applying SIMD to Scalar Code

Implementing Abs with Logic

- Reminder: SIMD FP format



- Code

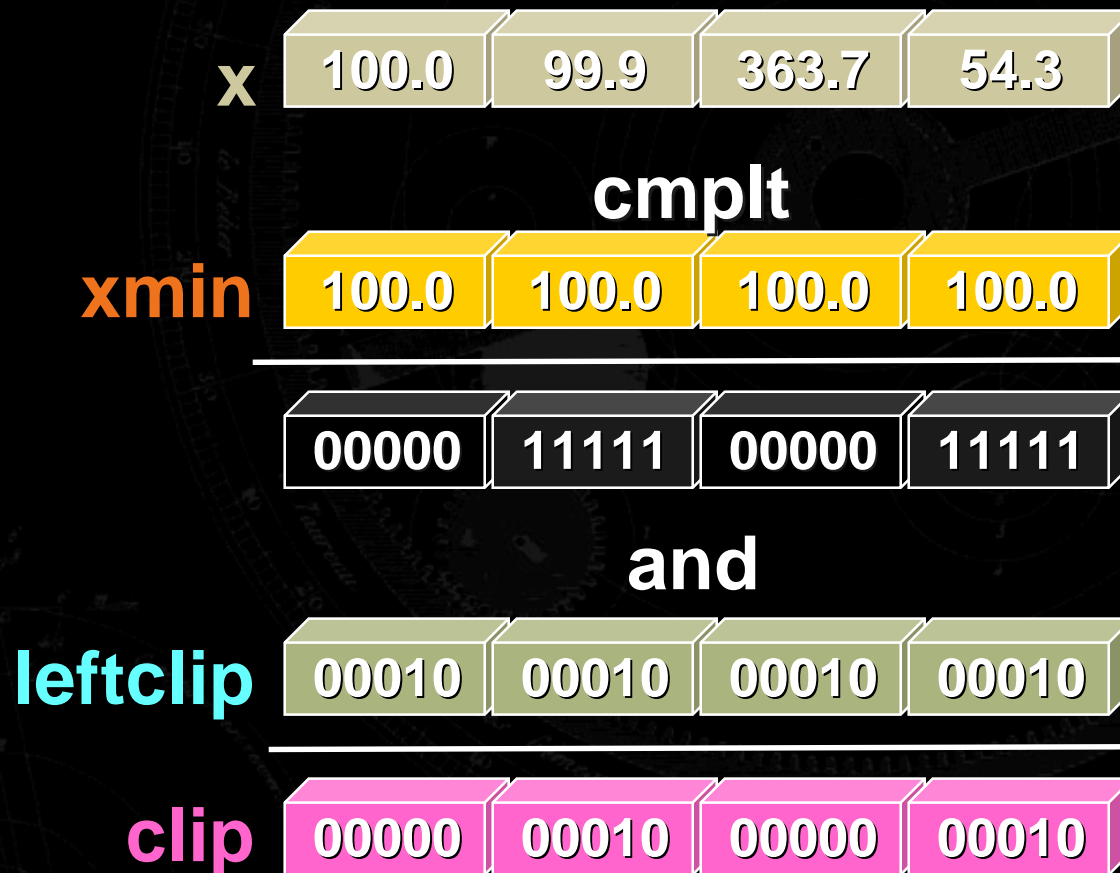
```
// r = abs(a)
CONST_INT32_PS(smask,
    ~(1<<31), ~(1<<31), ~(1<<31), ~(1<<31));
r = smask & a;
```

Flag Accumulation: Original Scalar Code

```
char clip = 0;

if (v->x < xmin)
    clip |= LEFTCLIP;
else if (v->x > xmax)
    clip |= RIGHTCLIP;
if (v->y < ymin)
    clip |= TOPCLIP;
else if (v->y > ymax)
    clip |= BOTTOMCLIP;
```

Flag Accumulation with SSE / SSE2



Flag Accumulation with SSE / SSE2 Code

```
DEFCONST_INT_PS(leftclip, LEFTCLIP);
... // DEFCONST for rightclip, topclip, botclip
F32vec4 clip, mask;
__m128i iclip;
unsigned uclip;

mask = cmplt(sx, ps_xmin);
clip = mask & leftclip;
mask = cmpgt(sx, ps_xmax);
clip |= mask & rightclip;
mask = cmplt(sy, ps_ymin);
clip |= mask & topclip;
mask = cmpgt(sy, ps_ymax);
clip |= mask & botclip;
// pack int32 → int8
iclip = (__m128i&)clip;           // cast type
iclip = _mm_packs_epi32(iclip, iclip); // pack 32 → 16
iclip = _mm_packus_epi16(iclip, iclip); // pack 16 → 8
uclip = _mm_cvtsi128_si32(iclip); // move to int
```


Test Passed Counting



```
static const int bitcount[16] = {  
    0, // 0 == 0000  
    1, // 1 == 0001  
    1, // 2 == 0010  
    2, // 3 == 0011  
    ...  
    4 // 15 == 1111  
};  
F32vec4 mask = cmplt(a, b);  
npassed = bitcount[move_mask(mask)];
```

Bonus Foils

Using Automatic Vectorizer
More Conditional Code with SIMD
Applying SIMD to Scalar Code

Applying SIMD to Scalar Code

SSE can be applicable inside a scalar algorithm
without global parallelization

Accelerate general processing with SSE operations

- SSE registers are more efficient than x87
- SSE divide, square root - rcp, rsqrt
- type conversions - cvtsi2ss, cvt(ss)2si...
- comparisons - comiss, comisd

Accelerate operations common to all
heterogeneous components

Is It Really-Really Scalar?

- In most cases, can easily load scalar data into SSE/SSE2 operands
 - Load four random 3-comp vectors:

```
void XYZToF32vec4(F32vec4& x, y, z, const XYZ* p0, p1, p2, p3)
{
    CONST_INT32_PS(m20, 0,~0,0,~0); // mask for elements 2, 0
    F32vec4 a, b, c, d, e;

    a = _mm_loadu_ps(&p0->x); // --,z0,y0,x0
    b = _mm_loadu_ps((&p1->x) - 1); // z1,y1,x1,--
    c = (m20 & a) | andnot(m20, b); // z1,z0,x1,x0
    b = (m20 & b) | andnot(m20, a); // --,y1,y0,--

    a = _mm_loadu_ps(&p2->x); // --,z2,y2,x2
    d = _mm_loadu_ps((&p3->x) - 1); // z3,y3,x3,--
    e = (m20 & a) | andnot(m20, d); // z3,z2,x3,x2
    d = (m20 & d) | andnot(m20, a); // --,y3,y2,--

    x = _mm_movelh_ps(c, e); // x3,x2,x1,x0
    z = _mm_movehl_ps(e, c); // z3,z2,z1,z0
    y = _mm_shuffle_ps(b, d, _MM_SHUFFLE(2,1,2,1)); // y3,y2,y1,y0
}
```

Avoiding SIMD Catches for Scalar Data

Example: load XYZ vector as SSE operand

Catch 1: Misalignment

```
● F32vec4 v; XYZ* vec; ●  
● x = _mm_loadl_pi(&vec->x); ●  
● v = _mm_movelh_ps(x, _mm_load_ss(&vec->z)); ●
```

- loadlo, loadhi slow when not 8-byte aligned

Catch 2: FP "Junk" data

```
● x = _mm_loadu_ps(&vec->x); ●
```

- Junk data leads to "special" values in math operations → slowdown!

Loading XYZ Vector as SSE Operand, Good Way

```
● F32vec4 v;  
● XYZ* vec;  
●  
● v = _mm_loadu_ps(&vec->x);  
● v = v & mask210;  
● // OR:  
● // v = _mm_shuffle_ps(v, v,  
● // _MM_SHUFFLE(2,2,1,0));
```

- One slow unaligned load, one logic
- Junk data masked out
- Aligned load would be much faster
- Data alignment is still important!

Loading XYZ Vector as SSE Operand, Better Way

```
● F32vec4 v, y, z;
● XYZ* vec;
● v = _mm_load_ss(&vec->x); // 0,0,0,x
● y = _mm_load_ss(&vec->y); // 0,0,0,y
● z = _mm_load_ss(&vec->z); // 0,0,0,z
● v = _mm_movelh_ps(v, y); // 0,y,0,x
● v = _mm_shuffle_ps(v, z, S(2,0,2,0));
```

- Three fast loads, two shuffles
- ~1.3x faster than non-aligned SIMD load
- ~2x slower than aligned SIMD

SIMD Element Sumup

- Used widely in SIMD-for-Scalar code
- Requires two sequential shuffles

```
● inline F32vec1 sumup(F32vec4 x) ●  
● { ●  
●     x += _mm_movehl_ps(x, x); ●  
●     ((F32vec1&)x) += _mm_shuffle_ps(x, x, S(3,2,1,1)); ●  
●     return x; ●  
● } ●
```

Parallel Element Sumups

Four element sumups in parallel

```
● inline F32vec4 sumup(F32vec4 a, b, c, d) ●
● { ●
●     a = unpack_low(a, b) + unpack_high(a, b); ●
●         // b3+b1, a3+a1, b2+b0, a2+a0 ●
●     c = unpack_low(c, d) + unpack_high(c, d); ●
●         // d3+d1, c3+c1, d2+d0, c2+c0 ●
●     b = _mm_movelh_ps(a, c); ●
●         // d2+d0, c2+c0, b2+b0, a2+a0 ●
●     d = _mm_movehl_ps(c, a); ●
●         // d3+d1, c3+c1, b3+b1, a3+a1 ●
●     a = b + d; ●
●     return a; ●
● }
```

Vector Normalize, SSE SIMD

- Element sumup considered earlier
- 5 shuffles, 2 multiplies
- Aligning data would speed it up!

```
● F32vec4 v, s;
● F32vec1 t;
● v = _mm_loadu_ps(inVec);
● v = v & mask210;
●
● s = v * v;
● t = sumup3(s); // sum up 3 lower elements only
● t = rsqrt_nr(t); // SSE scalar
● v *= _mm_shuffle_ps(t, t, S(0,0,0,0));
●
● _mm_storeu_ps(outVec, v);
```

Vector Normalize, SSE Scalar

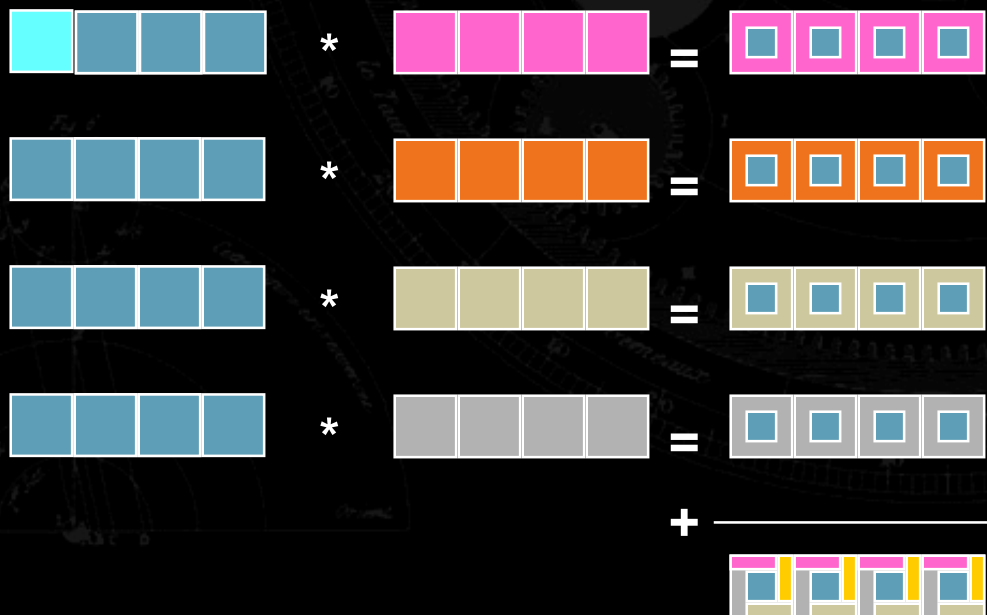
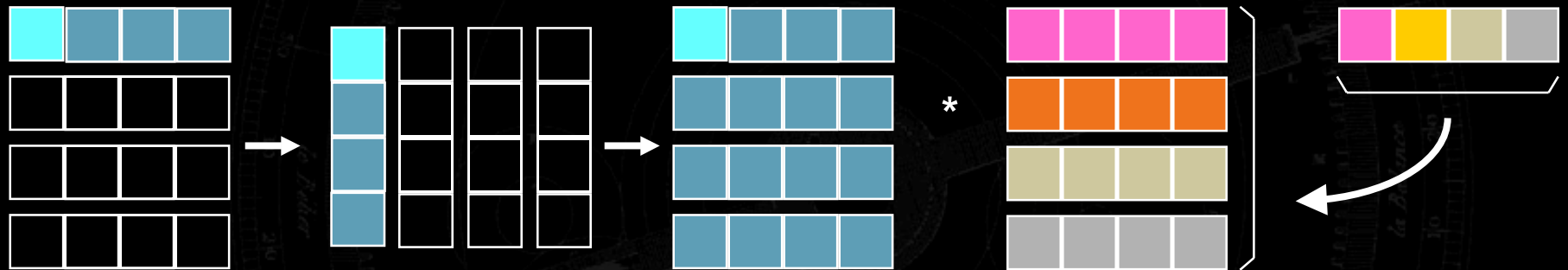
- 0 shuffles, 6 multiplies, reschedulable loads
- ~20% faster than unaligned SSE SIMD
- ~15% slower than aligned SSE SIMD

```
● F32vec1 x, y, z, t;
● x = _mm_load_ss(&inVec->x);
● y = _mm_load_ss(&inVec->y);
● z = _mm_load_ss(&inVec->z);
● t = x * x + y * y + z * z;
● t = rsqrt_nr(t); // SSE scalar
● x *= t;
● y *= t;
● z *= t;
● _mm_store_ss(&outVec->x, x);
● _mm_store_ss(&outVec->y, y);
● _mm_store_ss(&outVec->z, z);
```

SSE SIMD or SSE Scalar?

- Depends on memory loads to processing operation ratio
- Aligned load / store are the fastest
- Homogeneous component processing is faster with packed operations
- Load / store of separate components is more efficient than unaligned SIMD load

Matrix by Vector, Smart Scalar



SIMD for Scalar Code Summary

- Inherently scalar algorithms also benefit from SSE / SSE2
- Aligning data is still important!
- Do not tolerate junk data elements
- Use SSE / SSE2 fast operations: reciprocal, compares, conversions

SIMD Synonyms

Nouns:

- SSE = SSE / SSE2 = SSE2 = SIMD
- "SIMD operand consists of {four} elements"

Adjectives:

- SIMD data = vectorized data = packed data

Verbs:

- SIMDize = vectorize = parallelize

Afternoon Break



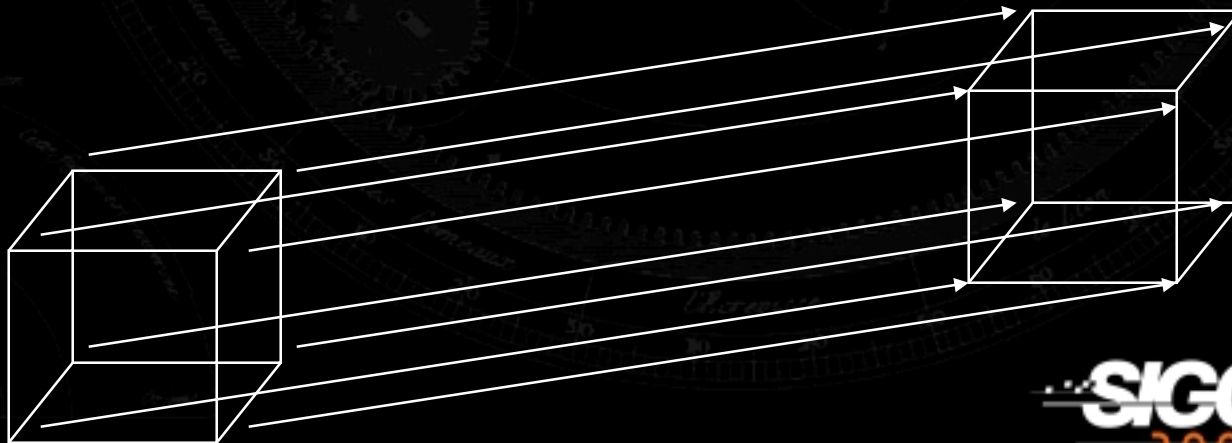
Architecture: SIMD computation & the memory game

Haim Barad
Staff Engineer
Intel Corporation

barad@acm.org

SIMD Computation

- Single Instruction, Multiple Data
- Almost all architectures have it
- Useful for vector style computation
 - E.g. Transform four vertices in parallel

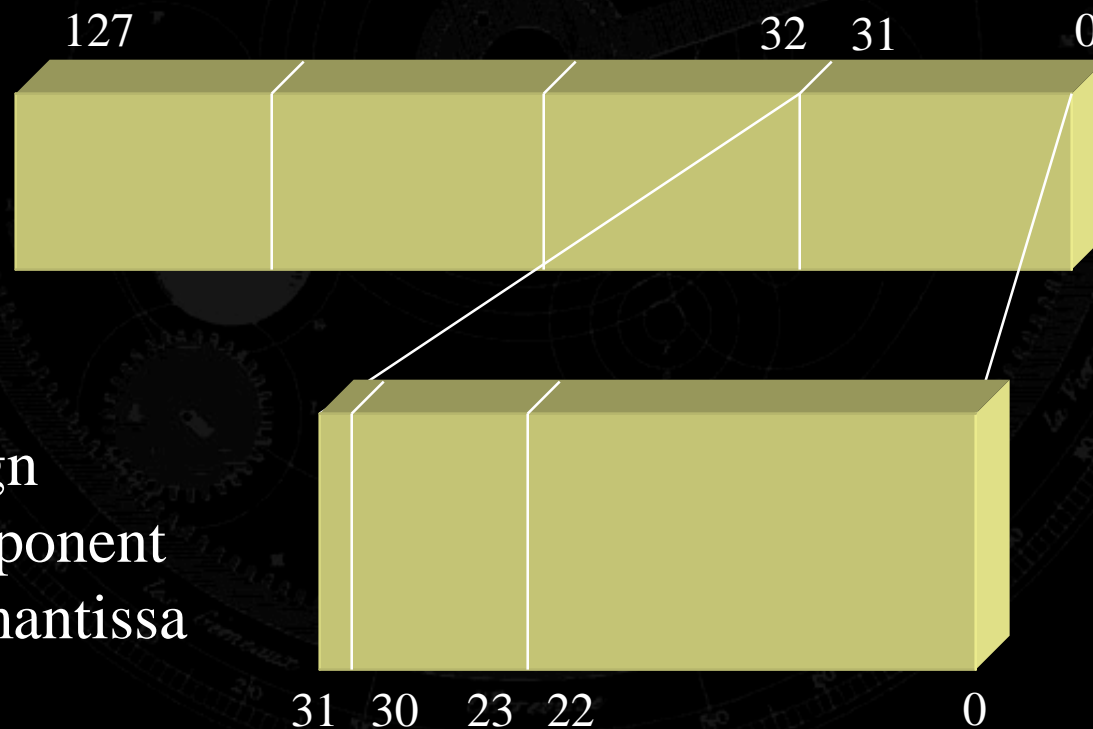


Streaming SIMD Extensions*

Registers

Eight 4-wide
Single-Precision
FP registers

1-bit sign
8-bit exponent
23-bit mantissa



Packed Operations

op is one of

- addps
- subps
- mulps
- divps

xmm0

a3

a2

a1

a0

xmm1

b3

b2

b1

b0

xmm0

a3 op b3

a2 op b2

a1 op b1

a0 op b0

Horizontal or Vertical?

Horizontal - find parallelism within the computation

Vertical - use serial task sequence, but operate on multiple data points

Vertical is often preferred

Dependencies limit instruction parallelism

- Code development from serial code is easier

Modify data structures to support vertical parallelism have better cache locality

- “Structure of Arrays” versus “Array of Structures”

SOA versus AOS

- **AOS (Array Of Structures)**

X0, Y0, Z0, Nx0, Ny0, Nz0, U0, V0,
X1, Y1, Z1, Nx1, Ny1, Nz1, U0, V0, ...

*Better cacheability
Better SIMD calculations*

- **SOA (Structure of Arrays)**

X0, X1, X2, X3, ... Y0, Y1, Y2, Y3, ... Z0, Z1, Z2, Z3, ...

Matrix Vector Multiply

Typical 3D operation

Load values in SOA format

- xxxx..., yyyy..., zzzz...

Follow with multiply and add operations

```
movaps    xmm0, [list+X+ecx];load X components
```

```
movaps    xmm2, [list+Y+ecx];load Y components
```

```
movaps    xmm3, [list+Z+ecx];load Z components
```

```
movaps    xmm1, [esi+m00]    ;m00  m00  m00  m00
```

```
movaps    xmm4, [esi+m01]    ;m01  m01  m01  m01
```

Matrix Vector Multiply (cont.)

Accumulate results...

We've just done four dot products in parallel!

```
mulps    xmm1, xmm0      ;x*m00 x*m00 x*m00 x*m00
mulps    xmm4, xmm2      ;y*m01 y*m01 y*m01 y*m01
addps    xmm4, xmm1      ;add the 2 results
movaps    xmm1, [esi+m02];load matrix element m02 (x4)
mulps    xmm1, xmm3      ;z*m02 z*m02 z*m02 z*m02
addps    xmm4, xmm1      ;add results
addps    xmm4, [esi+m03];add last element of matrix
```

Loop back to pick up next 4 vertices...

The Memory Party Game



If you can't bring data in fast enough or spit it out fast enough...

- SIMD is of little or no use
- You are now limited by bandwidth!

Invite the data ahead of time...

Prefetch - hides latency by bringing in data before you need it

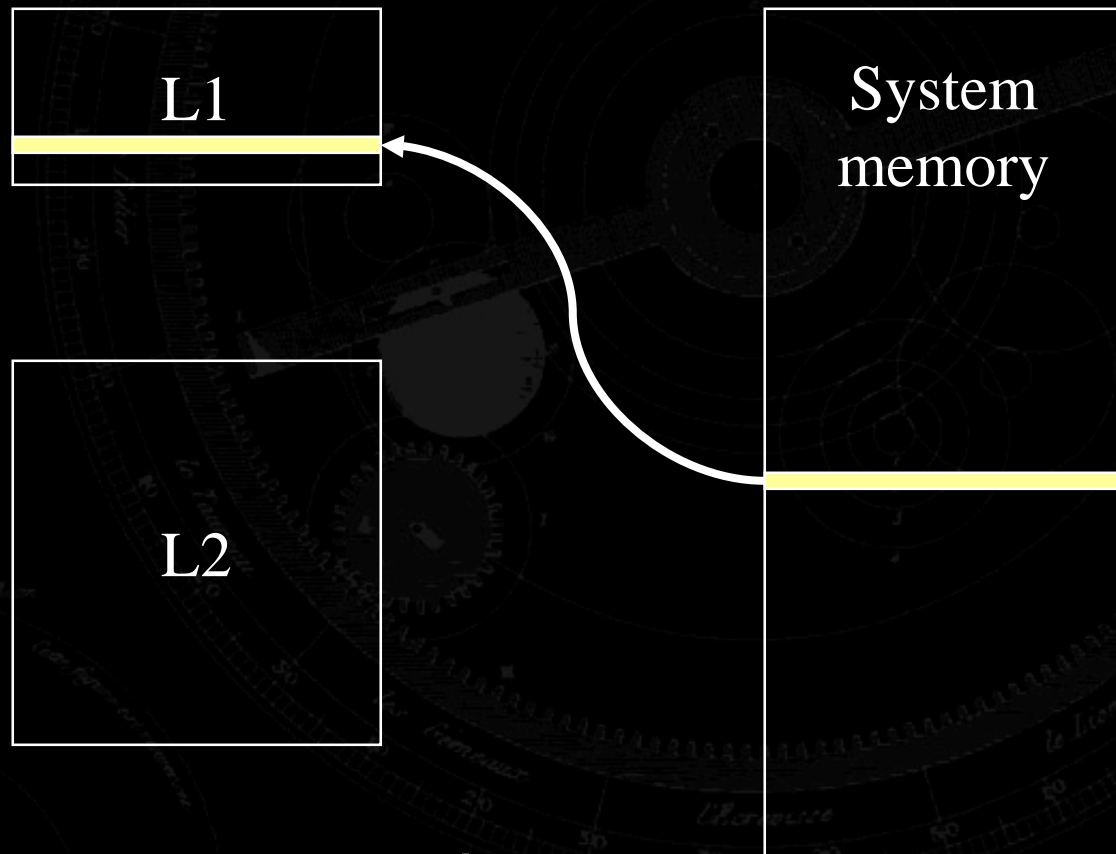
Cache latency still exists

- Be careful to prefetch far enough in advance

Watch resource limitations

- Prefetching too much can thrash the cache
- Limitations on number of fill buffers
 - Six in recent Pentium® III Processors
 - Over 20 in Pentium® 4 Processors

Prefetch illustrated



`prefetchnta [esi]`

Prefetching data...

loop

```
movaps xmm1, [edx + ebx]
```

```
movaps xmm2, [edx + ebx + 16]
```

```
;Prefetch next iteration data into cache
```

```
prefetchnta [edx + ebx + 32]
```

```
; ... perform calculations on this iteration...
```

```
; ...
```

```
add ebx, 32
```

```
cmp ebx, buff_size
```

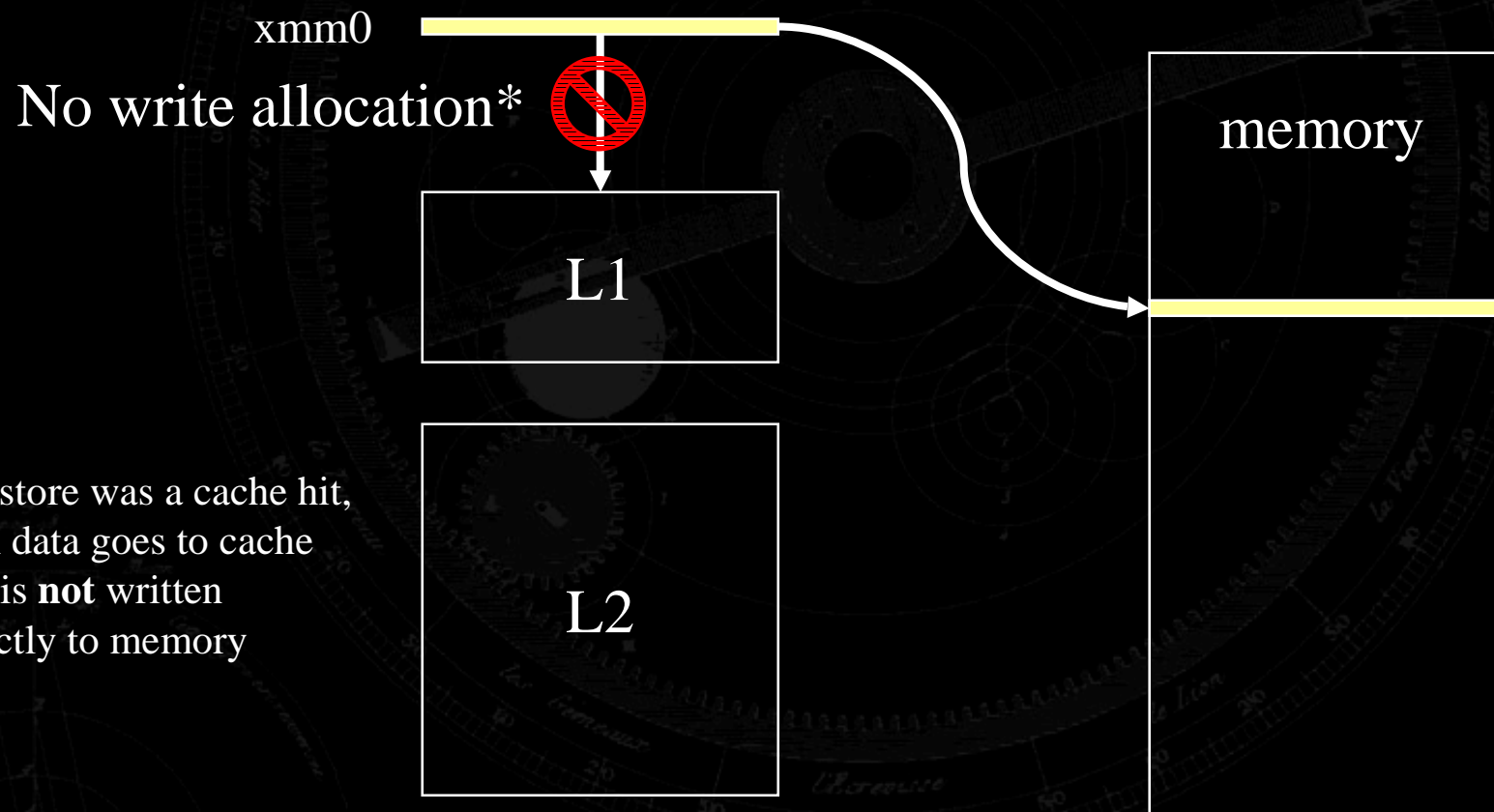
```
j1 loop
```

Quick exit from the party...

Streaming stores - stream output data directly to system memory without updating caches

Or, write directly to non-cachable memory such as AGP

Streaming store illustrated



* If store was a cache hit,
then data goes to cache
and is **not** written
directly to memory

`movntps [esi]`

Optimizations Strategy Lab

Haim Barad
Staff Engineer
Intel Corporation

Haim.Barad@intel.com

Lab overview

A simple 3D geometry engine

- Pipeline architecture
- SIMD implementation

Analysis

- Time and event based sampling
- Optimizations to further improve performance

Lab files and options

Main.cpp has options for different pipes

- Scalar vs. SIMD
- Single-pass vs. Multipass
- SOA vs. AOS

Prefetches can be enabled for SIMD pipe

- Put prefetches in soa_SP_SIMD_pipe.cpp

Analysis steps

Run app in VTune under TBS

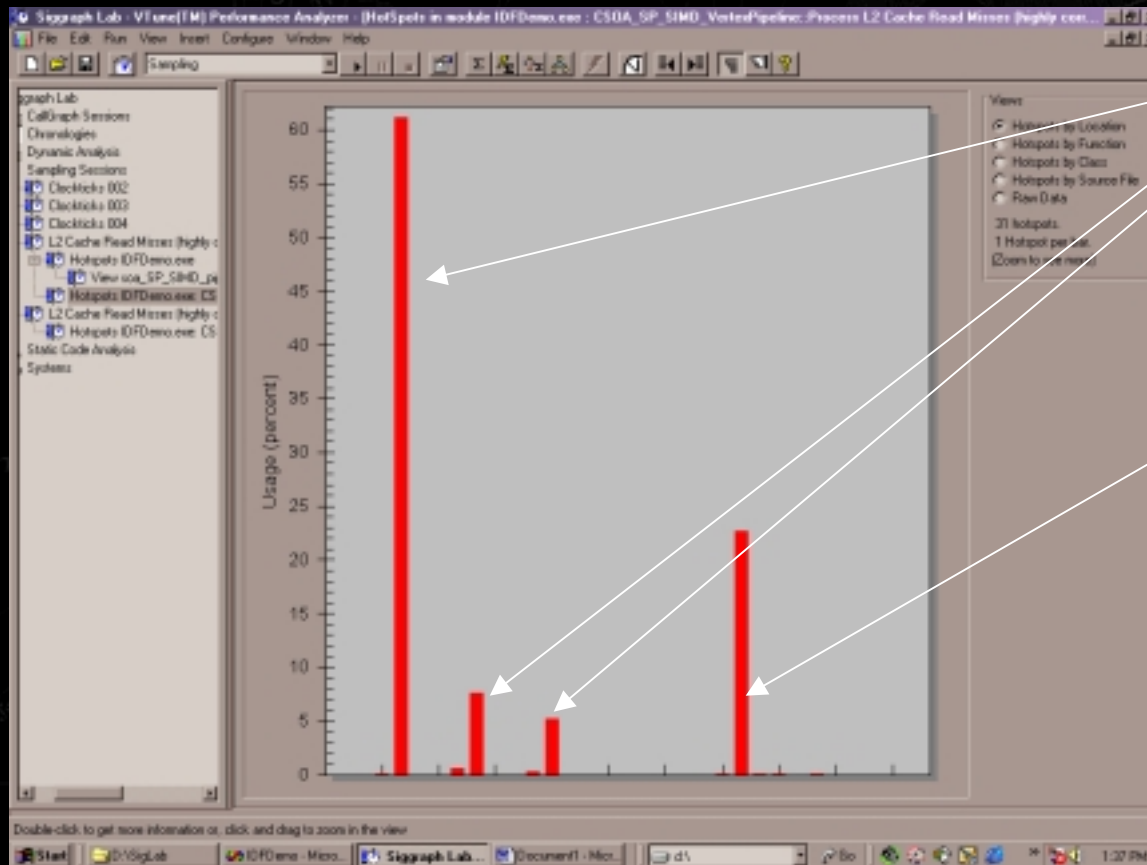
- Notice hot spots in the profile results

Track down causes

- Cache misses are likely in this app
- Measure with EBS (L2 misses)

Prefetch optimization

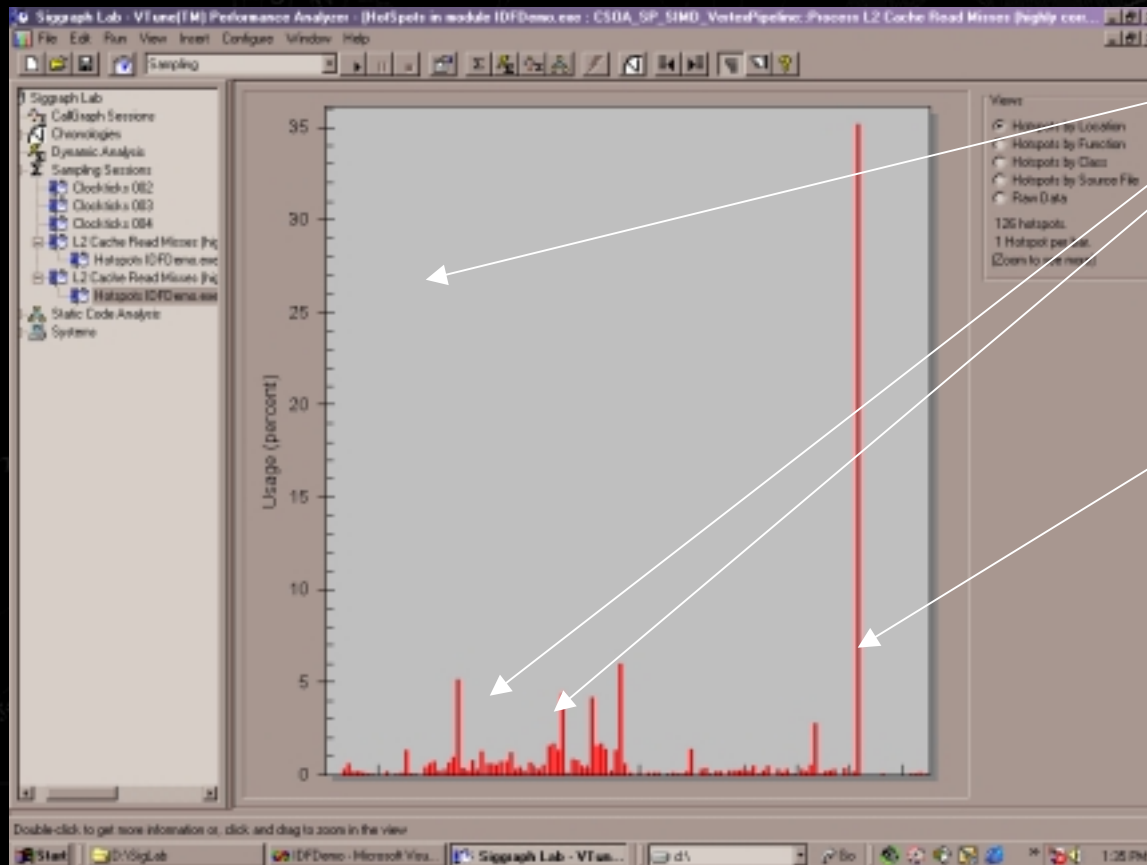
L2 Cache misses: SIMD pipe before prefetch optimization



Cache misses on
position loads

Allocate for
writing results

L2 Cache misses: SIMD pipe after prefetch optimization



Load latency
is gone!

We still have
allocate for
writing results

Processor & Platform Analysis Tools

Dean Macri
Technical Marketing Engineer
Intel Corporation

dean.p.macri@intel.com

What We'll Talk About Today...

- Platform performance issues
- Optimizations 101
- Finding the *Hot Spot*?
- Analysis techniques
- Try it out! (lab)

Platform Performance Issues

Software increasingly needs to be multi-platform

- You don't always develop on your target machine!

Platforms vary in many ways

- CPU horsepower, ISA, memory, graphics adapter

Optimizations should be focused where they can add the most value!

- Once you've got a solid algorithm, optimize to the target platform

Typical Excuses of Sub-Optimal Performance

1. *Not My Code!*
2. *Drivers*
3. *OS*
4. *Compiler*
5. *Processor*

Typical Causes of Sub-Optimal Performance

Poorly written code

- Function calls for small, high frequency routines

Inefficient use of data

- Bad cache coherence, data alignment, etc.

Poorly matching code to underlying platform / processor architecture

'gotchas' in specific architectures

Find the bottleneck!

Optimizations 101

Step 1: The Benchmark

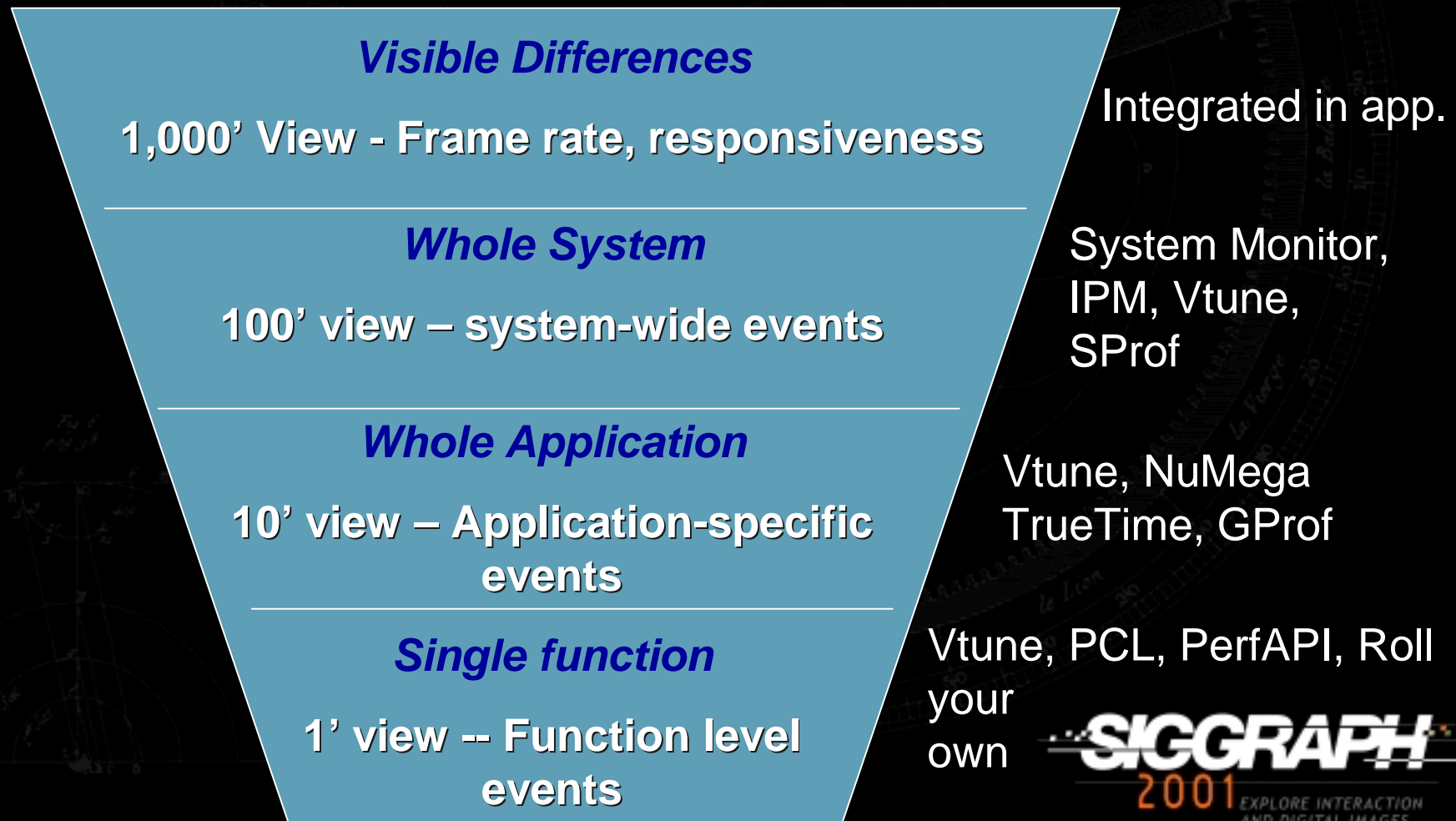
- Run the benchmark and record the performance.

Step 2: Find the hot spot

- Run the benchmark using a profiler to identify which areas of the application are consuming the most time.

Step 3: Optimize and verify the improvement

Use the Funnel Approach



1000' View Monitoring

19 fps (clamped at 30 fps) — katmai enabled
34386 triangles processed
16357 triangles rendered
crew 1 frame 360
camera 1 camera mode 7
control mode 0 control logic 0
125m visibility 5 lights
22530K free for textures (from 29/11K)

Frames / Sec

Triangles Rendered /
Frame

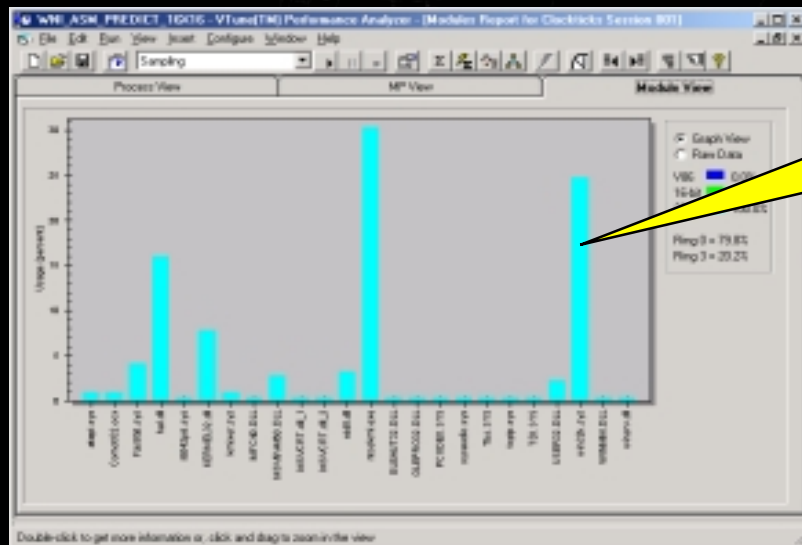
Lights,
Visibility

Texture
Usage

Other Metrics:

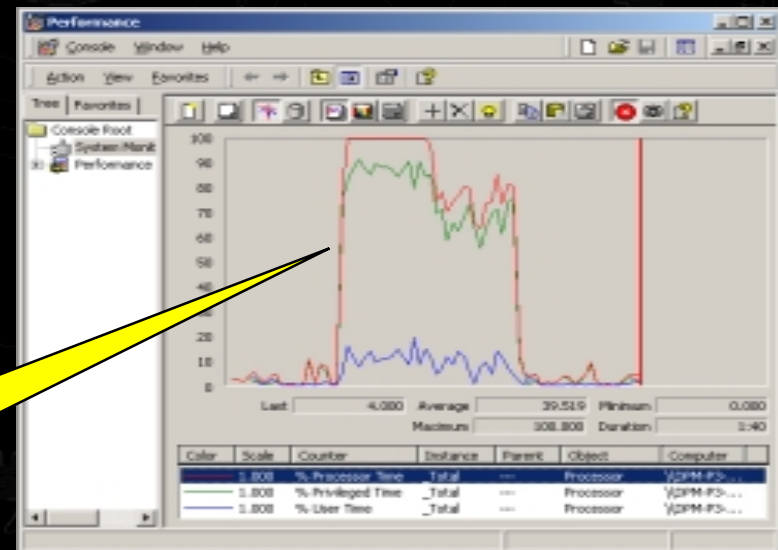
- Object count
- Memory
- etc.

100' View Monitoring

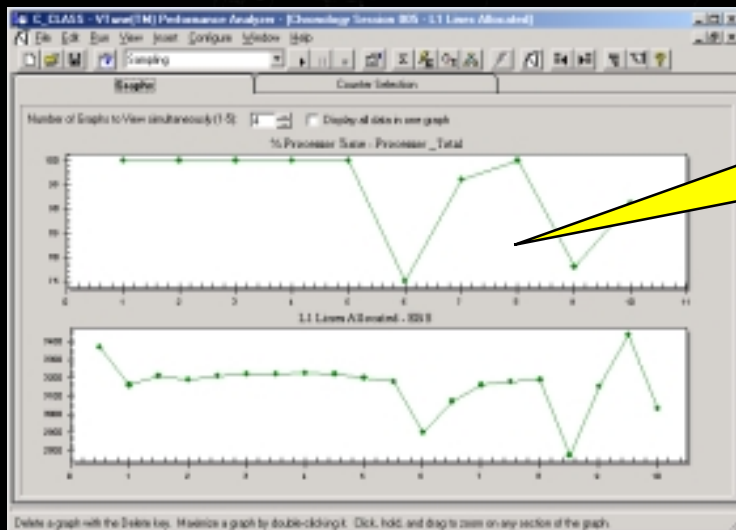


Vtune - Shows all the software running in the system. Larger bars are consuming more time.

System Monitor --
Displays events for
various objects over time.

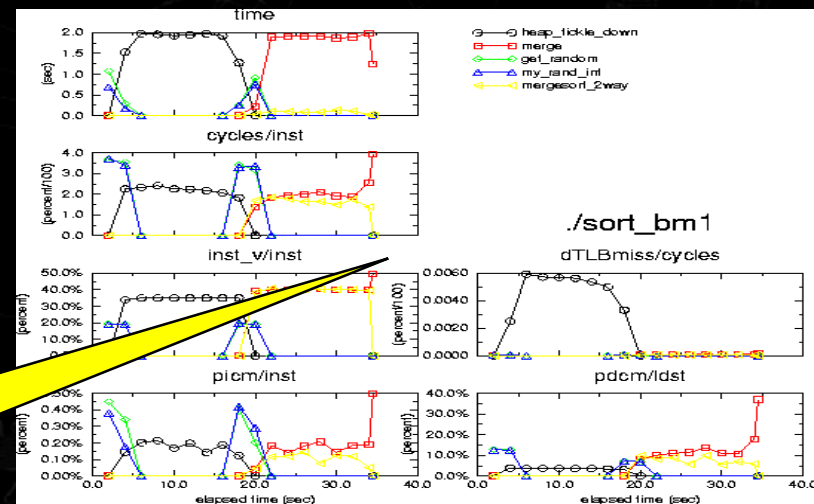


100' View Monitoring



Vtune - Chronologies displays events over time like System Monitor.

Sprof - Unix/Linux tool for profiling the system and applications



3rd Party Hardware Counters for Chronologies

System Monitor supports 3rd party counters

- Special purpose .DLLs for accessing hardware

VTune is extensible by 3rd parties

- Simple SDK available for creating .DLLs

Graphics IHV's can use to optimize titles for their hardware

- Give performance .DLL to title ISV
- ISV uses VTune/SysMon to get max performance

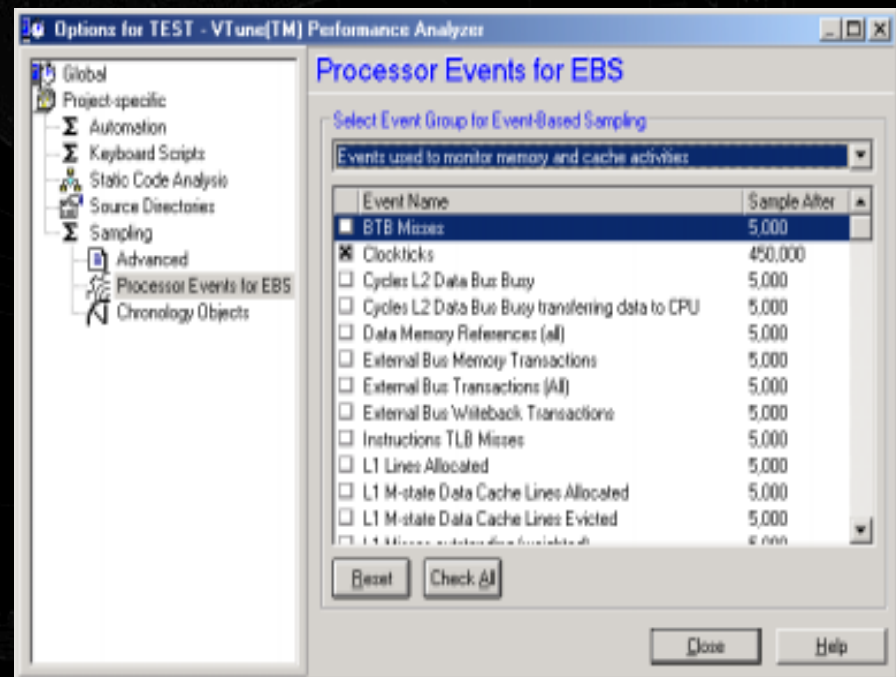
10' View Monitoring

There are profiling counters built into most processors
Can sample on more than just time!

- L1 and L2 cache misses, Branch mispredictions, partial register stalls

Can also compute ratios

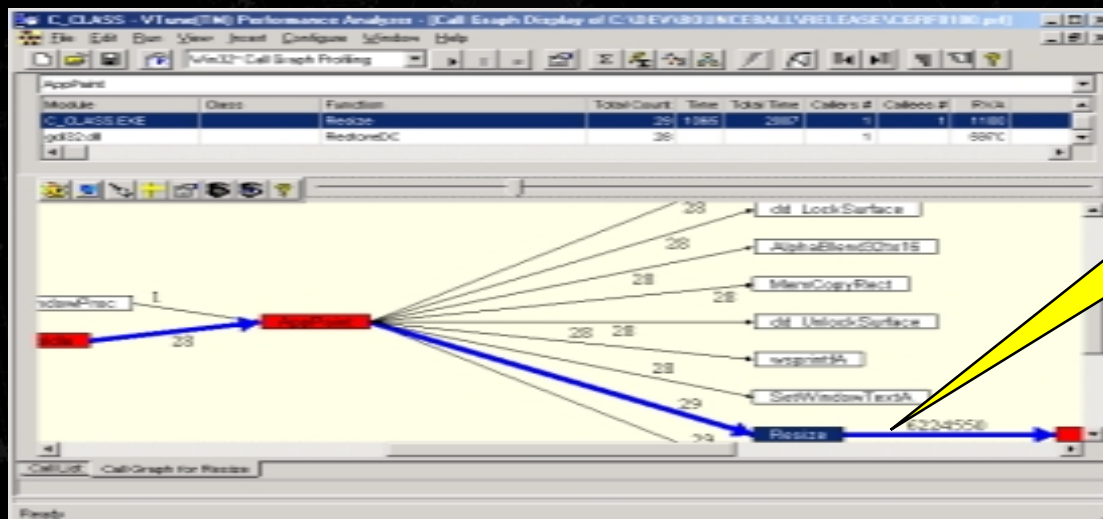
- clocks per instructions, bus utilization, data access versus cache misses



Call Graph Profiling

Goal is to get function call graph and call list

- Parents and children
- Time for itself and itself + all children
- Find the critical path



Critical path is blue

1' View Monitoring

Roll your own

- Write a privileged driver
- Instrument your code

Use a pre-rolled tool

- IPM
- Vtune API
- PCL - Performance Counter Library
- PerfAPI

Controlling Profiling

Add code to configure, start, and stop profiling within your app

- specific to VTune, IPM, PCL, PerfAPI, etc.

Rebuild your app and run it to collect data

Use a GUI or database viewer to analyze the results

The Start, Stop, and Config APIs

Profile code without the GUI

Data file is generated which is imported into VTune for analysis

Great for testing multiple configurations

Include VTuneApi.h, link VTuneApi.lib

```
int VtStartSampling(comment string);  
int VtStopSampling (void);  
int VtConfigSampling(options, interrupt type,  
                    SamplingIntervalInMicroSec, SampleBufferSizeInBytes,  
                    MaxSamples, Duration, StartDelay, SamplesFileName,  
                    EventCode, EBSSampleAfter);
```

Questions?



SIGGRAPH
2001 EXPLORE INTERACTION
AND DIGITAL IMAGES

Try It Out - Labs

Optimizations for Handheld & Embedded Devices

Haim Barad
Intel

Handheld platforms (PDAs)

PDAs (e.g. Palm Pilot, Compaq iPaq)

Embedded CPUs = low power \Rightarrow battery life!

Performance is now an important factor

- Video
- 2D players (e.g. video and Flash)
- 3D apps/games
- Speech recognition

Performance factors

This class of chips (e.g. StrongARM)

- Have no floating point hardware!
- Have no integer divide!
- Smaller caches than desktop CPUs
- Multiply units – “early out” (small finishes faster)

Performance Roadmap

Intel XScale

- Ramp to 1GHz (demonstrated at Spring IDF 2001)
- New architectural features
 - MAC w/40-bit accumulator
- Manufactured on advanced process
 - High frequency, low power!

Platform factors

Small displays (320x240)

- Smaller textures are required (mip mapping)
- Lower resolution models are ok
- Landscape mode has linear memory access

Small depth of display (up to 16-bit)

- Reformat and resize textures

Optimizations Steps

No floating point - CRITICAL STEP!!!

- Emulation is VERY slow (~50x)
- Use fixed point instead
- Smaller operands finish faster than larger
- Be careful of precision and dynamic range

Optimizations Steps (cont)

Be careful of cache sizes

- Current StrongArm has 16K/8K (I/D) caches
- XScale will have 32K/32K/2K (I/D/mini-D) caches

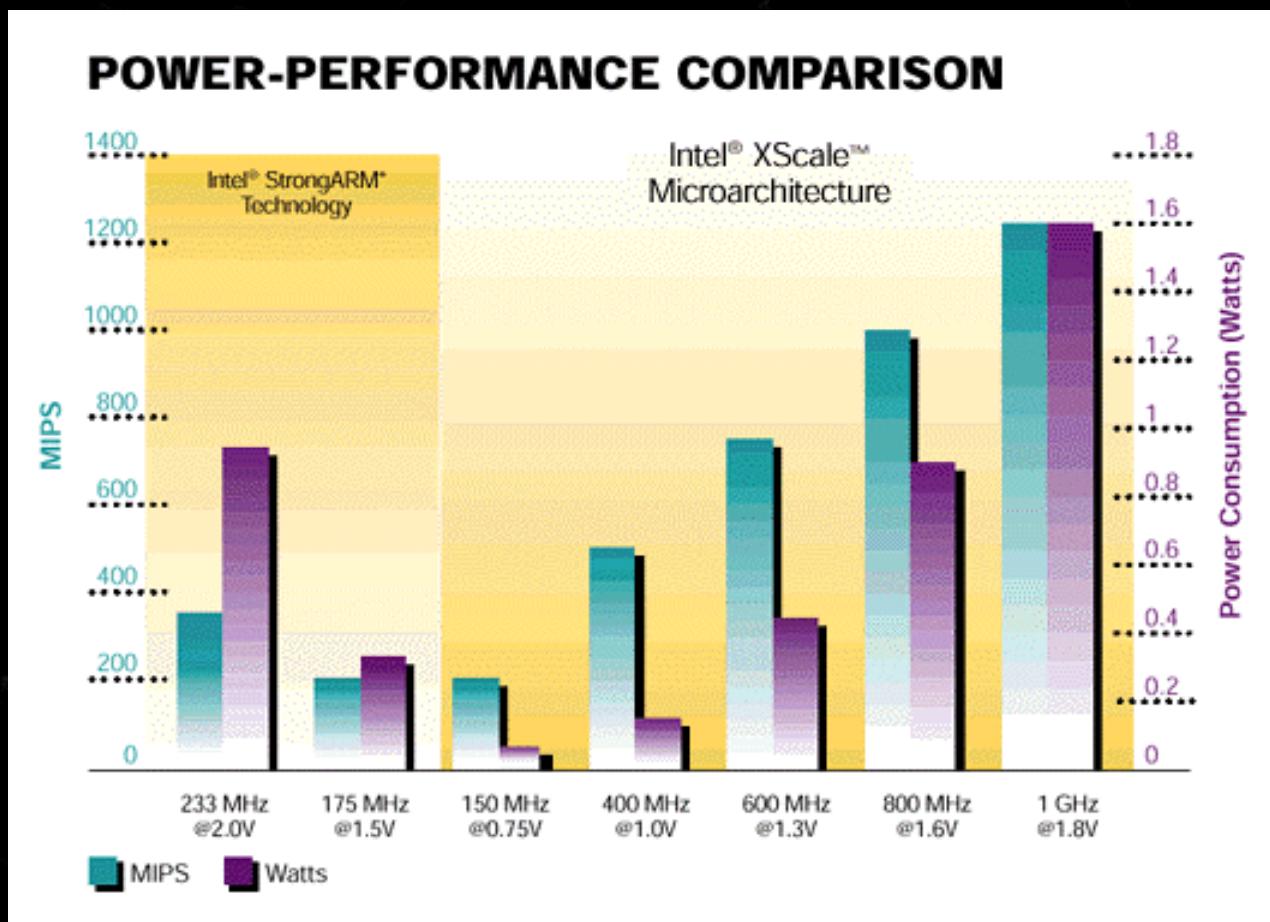
Be careful of compiler generated code

- Tools are not yet as mature as desktop versions
- Many OSs to support

Optimized libraries are available

- Intel's Integrated Performance Primitives (IPP)

Example of power and performance



A Small Demo

SIGGRAPH
2001 EXPLORE INTERACTION
AND DIGITAL IMAGES

Course Summary



Online References

A master link page for this course is at

- optimizations.org

Check for updates to materials!